VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Návrh, implementace a zavedení automatizovaných testů na straně klienta a serveru v robustním systému GX

Design, Implementation and Release of Automated Test Scenarios for GX Client and Server

2017 Bc. Ladislav Boštík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Ladislav Boštík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Návrh, implementace a zavedení automatizovaných testů na straně klienta a serveru v robustním systému GX
Design, Implementation and Release of Automated Test Scenarios for GX Client and Server

Jazyk vypracování: čeština

Zásady pro vypracování:

Výsledkem práce bude seznam zdokumentovaných a naimplementovaných testovacích scénářů vycházejících ze zadání GX na straně klienta a serveru.
Druhou částí bude navržení a realizace automatických testů nad celým robustním systémem GX.
1. Seznamte se s technologiemi systému GX na straně klienta a serveru.
2. Nastudujte technologie testování klienta a serveru dle použitých technologií systému GX a navrhněte způsoby a nástroje testování.
3. Dle zadání GX navrhněte testovací scénáře.
4. Implementujte scénáře.
5. Navrhněte způsob automatizovaných testů.
6. Zprovozněte automatizované testy.
7. Zdokumentujte postupy pro ostatní programátory pro vytváření testů.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Marek Bober**

Konzultant diplomové práce: Ing. Jan Martinovič, Ph.D.

Datum zadání: 01.09.2016

Datum odevzdání: 14.07.2017

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohašuji, že jsem tuto diplomovou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 28.6.2017

.....
podpis

Prohlášení firmy

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava

V Ostravě dne 28.6.2017

. . . . . .

podpis

GX SOLUTIONS BOHEMIA, s. r. o.
V Oblouku 114, 251 01 Čestlice
IČ: 26850010 • DIČ: CZ26850010
Tel.: +420 225 396 333
www.gxsolutons.eu

Rád bych poděkoval vedoucímu práce, panu Ing. Marku Boberovi ze společnosti GX Solutions Bohemia s.r.o. za velmi cenné rady, získané zkušenosti při vývoji a velice kvalitní vedení při návrhu a tvorbě modulu pro automatizované testování.

Abstrakt

Výsledkem práce bude seznam zdokumentovaných a naimplementovaných testovacích scénářů vycházejících ze zadání GX na stran klienta a serveru. Druhou částí bude navržení a realizace automatických testů nad celým robustním systémem GX. Kromě samotného nastudování systému GX-GO a nastudování technologií k modulu automatizovaného testování bude třeba korektně zakomponovat všechny požadavky do nového řešení. Modul bude hodnocen jako validní v případě, že bude schopen automatizovaných testů přesně podle specifikace.

Klíčová slova: klient, server, testování, testy

Abstract

Thesis result will contain list of documented and implemented testing scenarios based on GX client-side and server-side specification. Second part will be focused on design and realization of automated testing module over the entire GX system. Appart from learning about the GX-GO system and technologies for automated testing, all the requirements will be required to be implemented in the new solution. Automated testing module will be seen as valid, if it will perform automated testing procedures exactly as required.

Key Words: client-side, server-side, testing, tests

Obsah

Seznam použitých zkratek a symbolů

TDD	–	Test-driven Development
TFS	–	Team foundation server
SVN	–	Subversion
XML	–	Extensible Markup Language
WCF	–	Windows Communication Foundation
GPS	–	Global Positioning System
DLL	–	Dynamic-link library
MSTest	–	Microsoft test
Moq	–	Moq
RAM	–	Random-Access Memory
SDRAM	–	Synchronous Dynamic Random-Access Memory
CPU	–	Central Processing Unit
MB	–	Megabyte
GB	–	Gigabyte
ID	–	Identification

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Tato diplomová práce poskytuje detailní popis analýzy požadavků, návrhu a vývoje modulo pro automatizované testování v monitorovacím systému společnosti GX Solutions Bohemia s.r.o. . Jde o systém, na jehož vývoji se podílím již delší dobu a díky kterému jsem získal mnoho znalostí a zkušeností z oblasti analýzy, návrhu, vývoje a testování.

Společnost GX Solutions se pohybuje na trhu již mnoho let a v oblasti monitorování vozidel patří ke špičce se svém oboru. Systém GX-GO umožňuje snadné sledování vozidel, přidělování úkolů řidičům, monitorování aktivit vozidel, kontrolu nad zakázkami a jízdami, možnost plánování a mnoho dalších. Celkově je systém GX-GO asi čtvrtá verze softwarového řešení této společnosti a zavádí mnoho nových technologií a postupů, jejiž úkolem je maximální efektivita a spolehlivost systému jako celku.

Vývoj modulu automatizovaného testování byl zahájen z jednoho prostého důvodu a tím je zefektivnění systému jako celku a odstranění nutnosti provádět testy manuálně. Cíl je tedy takový, že programátor pouze napíše testovací případy a modul je sám při další iteraci zpracuje a vyhodnotí. Protože systém, o kterém se bavíme, je opravdu obrovský a neustále roste, jeden z požadavků mimo jiné byla absolutní dynamičnost a schopnost automaticky reagovat na přidávání nových testů, modulů nebo komponent.

Proto je modul samotný navržen jako autonomní část, která běží nezávisle na zbytku systému, ale zároveň dokáže získat ze systému cokoliv, co požaduje pro korektní provedení testů. Testy jako takové též nejsou něco, co můžeme vnímat jako jednotnou věc, minimálně je můžeme rozlišit na testování metod jádra systému, testování databáze a jejich metod a testování uživatelského rozhraní.

Důvodů pro vlastní řešení automatizovaných testů je více. I když na trhu existuje mnoho různých řešení jak pro všechny možné typy testů, tak pro celkovou automatizaci testovacího procesu, žádný se neukázal dostatečně komplexní. Tyto detaily budou popsány v sekci 2, kde bude několik úspěšných produktů pro testování rozebráno a evaluováno. Na konci této evaluace bude proveden celkový přehled řešení.

2 Cíle diplomové práce

Jak již bylo řečeno, záměrem této diplomové práce je vytvoření komplexního testovacího nástroje pro stále rostoucí monitorovací systém GX-GO. Modul pro automatizované testování tak bude fungovat jako autonomní součást celého systému a jeho úkolem bude testovat systém jako celek podle poslední dostupné verze, kterou programátoři publikovali. Protože tento modul poběží na serveru, kde nebude možnost interakce s uživatelem, musí být navržen tak, aby byl sám schopen provádět požadované úkoly, přizpůsobit se současným podmínkám a případně se vypořádat se vzniklými problémy.

Seznam cílů:

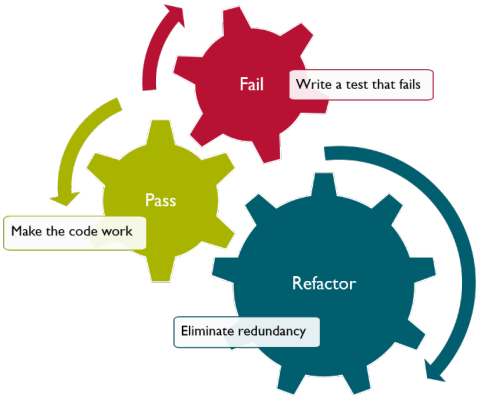
- Autonomní funkcionalita
- Stále aktuální
- Testování v případě změn
- Překládání projektů za běhu
- Kompletní kontrola chyb
- Regresivní testy
- Unit testy
- Mock testy
- Vlastní report formát

Pro samotné automatizované testování už na trhu existuje mnoho různých řešení, a to jak těch méně, tak i více efektivních. Každé řešení má nějaké své výhody, na druhou stranu ale má i slabiny, které byly ve finální fázi důvodem, proč dané řešení nebylo použito. Několik existujících řešení bude představeno níže.

2.1 Specifikace záměru

V rámci TDD (Test-driven Development) má tento modul ve výsledku zefektivnit jak samotnou práci programátorů, tak výkon a spolehlivost systému jako celek. Oproti existujícím standardům, kde je specifická osoba, která navrhuje testovací scénáře, tester, jehož práce je tvorba, spouštění a vyhodnocování samotných testů, a vývojář, který stojí za návrhem původního kódu a který je případně zodpovědný za jakékoliv nedostatky, zde se k řešení přistupuje trochu jinak.

Schéma výše poskytuje zjednodušený pohled na samotný TDD. Prakticky jde o tři fáze, které se opakují. V první fázi se tester pokusí navrhnout testy tak, aby určitě byly vyhodnoceny



Obrázek 1: Test-driven Development

jako chybné. Poté je potřeba odpovídajícím způsobem upravit kód, aby test mohl projít. V poslední fázi je pak třeba refaktorovat upravený kód, odstranit přebytečné nebo duplicitní části a všeobecně kód zefektivnit.

Záměrem modulu pro automatizované testování je spíše testování kódu v stále se rozvíjejícím prostředí. Programátor, který tak bude implementovat nové metody a funkcionality, zároveň bude psát i testy pro tyto metody. Tím odpadne nutnost toho, aby se těmito úkoly musel zabývat tester. Návrhy testů se budou řídit specifikacemi firmy a nebude tedy nutné provádět plánování, protože programátor napíše test pro každou přidanou funkcionalitu, kterou bude chtít zakomponovat do systému.

Také bylo zmíněno, že modul má pracovat s nejaktuálnější verzí systému, nikoliv s verzí, která běží na produkčních serverech. Všechny verze systému a všechny publikované změny od programátorů, jsou na firemních **SVN** serverech. Modul proto bude získávat poslední verze systému právě odtud. Tento způsob umožní testování funkcionality metod vzhledem ke změnám v systému. Tak tedy bude možné ověřovat, že nové změny a nové funkce nějak negativně neovlivňují již existující funkcionalitu, předejde se tak potencionálním problémům, právě díky automatizovanému modulu.

Proto se došlo k závěru, že pro monitorovací systém GX-GO bude efektivnější navrhnout separátní autonomní modul, kterému se umožní přístup na **SVN** servery a který bude schopen se o testování postarat sám, bez nutnosti využívat externí, již existující řešení. Tak bude možné modul sestavit tak, aby splňoval požadavky a díky nezávislosti na třetí strany nebude problém modul na základě případných požadavků případně rozšířit nebo modifikovat tak, aby splňoval nové požadavky a potřeby.

V následující sekci bude nicméně rozebráno několik komerčních řešení automatizovaného testování a u každého z nich budou vytaženy jak kladné tak záporné faktory.

2.2 Existující řešení

Komerčních řešení na trhu existuje mnoho. Pro průzkum je zvoleno několik možností nástrojů pro automatizované testování. Nástroje budou v následujících částech analyzovány, rozebrány a evaluovány. Z důvodu specifických požadavků společnosti je třeba nástroje porovnávat právě proti těmto požadavkům.

2.2.1 TestRail

První známý nástroj je *TestRail*, zdroj [1]. Jde o velmi pokročilý nástroj, který je schopen komplexního testování rozsáhlých softwarových řešení. Nicméně pro tento software je třeba využívat externí servery pro testování, a tento fakt je v rozporu se záměry společnosti GX Solutions. Také je tu ještě fakt, že *TestRail* se spíše specializuje na back-end testování, nicméně jedním z cílových požadavků modulu je i pokročilé testování UI.

TestRail používá pokročilé a responsivní webové rozhraní, které umožňuje přístup k testům buď přes cloud nebo server. Testování zde lze provádět jak manuálně, tak plně automatizovaně. Podporované typy testů jsou jak běžné Unit Testy, tak základní UI testů v rámci rozhraní. Report výsledků testů je zde psán ve velice komplexním formátu, kde není možnost vlastního stylování výstupu. Také zde není podpora verzování nebo lokálního překladu projektů z externího zdroje.

Pozitivní faktory:

- Efektivní testování přes cloud
- Rychlé testování na vlastním serveru
- Detailní webové rozhraní pro kontrolu
- Back-end testování velice rychlé
- Testování včetně kontroly databáze
- Testování služeb a interakce

Negativní faktory:

- Nepodporuje verzování
- Instalace na vlastním serveru je drahá
- Špatně přehledný report
- Není schopen překladu za běhu
- Nekontroluje spustitelnost a korektnost projektů

2.2.2 TFS

Další známý nástroj je *TFS* (Team Foundation server), zdroj [2]. Jde o nástroj společnosti Microsoft, který umožňuje verzování systému, management zdrojů, testování, publikování a mnoho dalších. V rámci *TFS* je možné jak verzování a překlad projektů na straně serveru. Tak se dá zajistit že projekty budou v korektním a použitelném stavu. *TFS* dále umožňuje pokročilé testování, včetně automatizace testů. Mezi podporované testy patří veškeré standardní testy, které jsou podporované v prostředí Visual Studio.

TFS se velice hojně používá, nicméně pro systém GX-GO není použitelný. Důvod je jednoduchý, systém a jeho verze jsou spravovány v rámci *SVN* serveru, který se nedá sloučit s funkcionalitou *TFS*. Dalším problémem by bylo zatížení na servery, protože *TFS* požaduje vlastní serverový prostor a licenční práva pro plné využití.

Pozitivní faktory:

- Podpora verzování
- Správa uživatelů s přístupy k systému
- Přímá integrace s vývojem
- Snadné a rychlé testování
- Podpora překladu projektů
- Kontrola spustitelnosti projektů

Negativní faktory:

- Nekomunikuje s SVN
- Potřebuje speciální server pro běh
- Příliš rozsáhlý report
- Nepodporuje modifikaci výstupu reportu
- Nepodporuje headless testování

2.2.3 Segron

Segron je společnost poskytující programové řešení pro komplexní end-to-end automatizované testy. Testovací modul je jen jednom z mnoha součástí tohoto řešení, dalšími je například Návrh a nastavení sítě, Podpora mobilních zařízení, Podpora vývoje software a Management projektů. Každý z uvedených modulů pracuje samostatně, není závislý na žádném jiném modulu. Moduly

jako celek tak mají za úkol zefektivnění a zjednodušení celkového vývoje software, ať už jde o malé řešení, nebo o komplexní systém s mnoha částmi.

Pro samotné testování je třeba nahrát projekty na server, kde již běží modul automatizovaného testování *Segron*. Na serveru tak potom může běžet jak celý systém, tak jeho testy. Nástroj společnosti *Segron* také poskytuje přistup do systému i v době testování, je tak možné ověřovat průběh testů, případně samotné testy sledovat.

Pozitivní faktory:

- Komplexní řešení pro podporu vývoje
- Specializace na back-end testy
- Poskytuje rychlé testování služeb
- Podporuje celou škálu testů
- Testování je velmi rychlé

Negativní faktory:

- Projekty musí být na serveru společnosti
- Testování pouze u firmy
- Nemá vlastní reporting
- Neefektivní UI testy
- Neumí headless testing
- Nepodporuje překlad za běhu

2.2.4 Telerik

Společnost *Telerik*, zdroj [4], poskytuje možnost komplexního řešení testů jak na straně klienta, tak na straně serveru. Pro použití samotného *Telerik* testu je třeba mít projekt, který sám o sobě už používá *Telerik* komponenty. Nástroj *Telerik* lze snadno integrovat do prostředí Visual Studio, kde je poté zapotřebí aplikovat licenci pro použití.

Konkrétně testovací modul *Telerik* vyžaduje kromě speciální licence ještě specifickou implementaci pro všechny testy na základě struktury předepsané společností *Telerik*. Tento fakt poněkud komplikuje integraci s již existujícími testy, protože tyto testy potom nemusí pracovat korektně nebo nemusí pracovat vůbec.

Pozitivní faktory:

- Efektivní rozšíření nad .NET
- Snadná integrace ve Visual Studiu
- Rychlé a efektivná testy
- Komplexní testování pro Telerik aplikace

Negativní faktory:

- Specifická specifikace testů
- Problémová integrace existujících testů
- Nepodporuje vlastní report
- Testování převážně pro Telerik aplikace
- Nutnost zavedení speciálního serveru pro testy
- Nepodporuje verzování

2.2.5 Belatrix

Nástroj pro testování společnosti *Belatrix*, zdroj [5], je vytvořen pro zefektivnění Agile vývoje software. Specializuje se na zjednodušení, komunikaci a feedback při vývoji a testování. Využívá dekompozici celkového řešení na menší logické celky, kde za pomocí prioritizace jsou jednotlivé testy seřazeny a poté evaluovány.

Pro samotné testování se po určení priorit definují ideální typy testů a jejich parametrů. Parametrizace zde může probíhat jak na základě předem naprogramovaných parametrů, tak na základě definovaných tabulek parametrů. *Belatrix* nástroj se zaměřuje hlavně na testování mobilních zařízení a back-end řešení. Pro mobilní zařízení se využívá nástroj *Soasta*, který mimo jiné slouží k zefektivnění samotného testování jako celku.

Pozitivní faktory:

- Detailní testování mobilních zařízení
- Komplexní řešení pro back-end testy
- Dekompozice systému
- Logická prioritizace
- Evaluace pro typy testů

Negativní faktory:

- Neadekvátní testy pro UI
- Pochybný report výsledků
- Neumožňuje testování služeb a interní komunikace
- Nedokáže tesotvat databázi

2.2.6 Rapise

Rapise software pro testování, zdroj [6], je rychlé a efektivní řešení pro testování a simulace akcí uživatele v rámci webových aplikací. Umožňuje širokou škálu funkcionalit pro testování, včetně komplexní definice samotných testů, kde tyto testy mohou být psány buď manuálně, nebo se dají automaticky generovat sledováním uživatelských akcí. Z těchto testovacích scénářů se poté generuje testovací skript, který se už dále používá pro samotné testování.

Mimo jiné zde existuje i jistá podpora pro unit testy, ale v tomto případě spíše ty unit testy, které jsou nějak svázané přímo s uživatelským rozhraním. Není tedy řešen samotný back-end testing. Kromě webových aplikací se *Rapise* zaměřuje i na testování interakcí pro aplikace na mobilních zařízeních a aplikace v desktop prostředí. Pro management samotných testů využívá nástroj SpiraTest, který umožňuje snadnou a efektivní kontrolu nad testy a jejich výsledky.

Pozitivní faktory:

- Komplexní UI testování
- Statické i dynamické generování test skriptů
- Zpětné přehrávání výsledků
- Cross-browser testy
- Data-driven testy
- Všeobecné testování rozhraní
- Umí headless testy

Negativní faktory:

- Vlastní test skripty
- Nepodporuje verzování
- Složité a rozsáhlé reporty
- Specializuje se na rozhraní
- Není schopen databázových testů

2.2.7 Testsigma

Testsigma, zdroj [7], je nástroj pro testování projektů a realizaci testů v prostředí cloud. Umožňuje speficikaci testů, specifikaci aplikací a požadavků. Plánování testů funguje na principu přiřazení test case k jednotlivým třídám testů a poté nastavení automatizace, kdy a jak se mají tyto testy spouštět. Test case je v tomto případě jakýkoliv typ testu, jako je Unit test, UI test a podobně. Pro veškeré definice poskytuje *Testsigma* přehledné rozhraní, které usnadňuje přehlednost jak samotných testů, tak výsledků.

Výsledky jsou produkované do výstupních reportů, které je možné do určité fáze modifikovat. Výstup reportu lze uživatelsky upravit, aby splňoval požadavky uživatele, jak má vypadat a co má obsahovat. Prostředí dále poskytuje konfigurace pro uživatele, kteří mají do systému přístup a kteří s ním mohou pracovat.

Pozitivní faktory:

- Všechno na jednom místě
- Cloud řešení
- Efektivní tvorba test case
- Možnost vlastního reportu
- Dynamický přehled výsledků
- Regresivní porovnávání

Negativní faktory:

- Nepodporuje překlad projektů
- Neumožňuje verzování
- Chybý kontrola korektnosti projektů
- Vyžaduje nahrání projektu na cloud

2.3 Porovnání řešení

V předchozích sekcích bylo rozebráno několik různých konkurenčních produktů určených k automatizovanému testování. Jak bylo vidět ve výsledných porovnáních u každého nástroje, vždycky existovaly kladné a záporné stránky. Jak bylo řečeno v sekci 2, v zadání modulu byly jisté požadavky, které bylo třeba zohlednit.

Každé z uvedených řešení má něco, čím se odlišuje od ostatních. Produkt *Rapise*, viz sekce 2.2.6, se například specializuje hlavně na UI testy a generování testovacích skriptů z akcí uživatele. Jde tak o velmi povedený simulátor pro například akceptační testy. Naproti tomu produkt

společnosti *Segron*, viz sekce 2.2.3, se specializuje na back-end testování, databázové testy a testy komunikace mezi službami.

Požadavek specifického report systému pro výsledky automatizovaných testů by byl třeba ideální produkt *Testsigma*, viz sekce 2.2.7, který mimo jiné umožňuje i detailní specifikaci průběhu testů a regresivního porovnávání výsledků. Problém ale je jeho požadavek, že projekt musí být na cloud serveru mimo prostředí zákazníka, v tomto případě společnosti GX Solutions.

Pro modul automatizovaného testování, tak jak je požadován, je třeba splnit všechny uvedené kritéria, jak již bylo zmíněno v sekci 2. Proto fakt, že existují produkty, které splňují třeba 70% požadavků, není relevantní. Specifické požadavky existují právě proto, protože výsledné řešení automatizovaných testů bude nutné zakomponovat už tak do velmi komplexního systému GX-GO.

Celkové shrnutí konkurenčních řešení je uvedeno v tabulce 1, kde jsou všechny analyzované produkty a u každého produktu je kontrolována určitá vlastnost, která je v zadání požadovaná. Pro akceptování externího řešení je tedy požadováno úplné splnění všech požadavků na modul automatizovaného testování.

Z uvedených produktů, jak je vidět, bohužel žádný neodpovídá detailně všem požadovaným specifikacím, proto bude třeba navrhnout vlastní řešení. V analytické části budou jednotlivé problémy a požadavky rozebrány a bude se řešit, jak tyto jednotlivé bude nutné vyřešit.

Tabulka 1: **Porovnání produktů**

	TestRail	TFS	Segron	Telerik	Belatrix	Rapise	Testsigma
Podpora SVN	Ne	Ne	Ne	Ne	Ne	Ne	Ne
Podpora verzování a vývoje	Ne	Ano	Ne	Ne	Ne	Ne	Ne
Překlad za běhu	Ne	Ano	Ne	Ano	Ne	Ne	Ne
Kontrola spustitelnosti	Ne	Ano	Ne	Ano	Ne	Ne	Ne
Unit testy	Ano	Ano	Ano	Ano	Ano	Ano	Ano
UI testy	Ano	Ano	Ne	Ano	Ne	Ano	Ano
Headless testy	Ne	Ne	Ne	Ne	Ne	Ano	Ne
Databázové testy	Ano	Ano	Ano	Ano	Ne	Ne	Ano
Vlastní report	Ne	Ne	Ne	Ne	Ne	Ne	Ano
Zpravování testů	Placený server, Cloud	Speciální server	Placený server	Kdekoliv	Kdekoliv	Kdekoliv	Cloud

3 Analýza

Výstupem z předchozí části, která se zabývala rozborem existujících řešení pro testování, je fakt, že žádné z uvedených řešení nesplňuje všechny body požadavků na modul. Z toho vyplývá nutnost vlastního návrhu řešení pro automatizované testování, aby byly splněny všechny body v požadavcích.

V analytické části budou popsány různé problémy a požadavky, které bylo nutno brát v úvahu, aby bylo dosaženo optimálního výsledku. Požadavků samotných bylo více, hlavně bylo nutné samotný systém celkově analyzovat, a na základě této analýzy popsat vlastnosti požadovaného řešení. Samotná analýza bude proto rozvržena do několika částí podle toho, čím se daná část zabývá.

V první části budou rozebrány jednotlivé části systému GX-GO, podle jejich důležitosti a hlavně podle jejich vlastností. Jednotlivé části budou popsány podle úrovně provázanosti a závislosti na ostatních částech systému.

Druhá část se bude zabývat tím, co je požadováno od samotného modulu pro automatizované testování. Hlavní zaměření bude celkově na požadavky, dále pak na funkcionalitu a výkon.

Finální část se zaměří na rozbor možných technických řešení pro modul automatizovaného testování. Řešit se bude jak samotná struktura modulu, tak jednotlivé části a funkce, za které budou jednotlivé části zodpovědné.

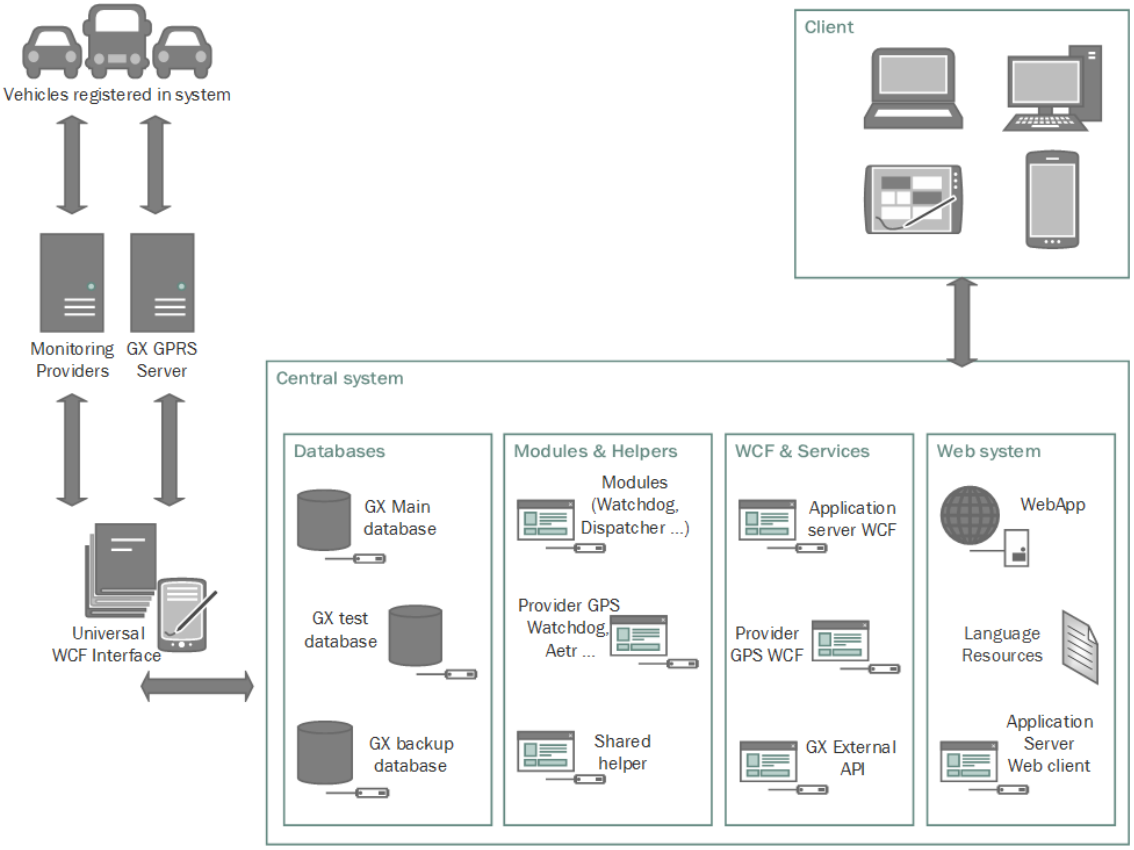
3.1 Analýza systému

Monitorovací systém GX-GO je složen z mnoha částí. Většinu z těchto částí lze vidět jako moduly, které převážně pracují samostatně. Samotné moduly jsou rozděleny podle toho, jaké mají řešit problémy. Jde například o moduly:

- AETR
- Watchdog
- Routing
- Report
- PrintForms
- Fleet
- Event Control
- Geocoding
- a další

Všechny moduly komunikují s databází a společnými knihovnami přes jednotné rozhraní. Hlavní komunikace vede vždy na centrální databázi, pokud tedy nutně některá část systému nevyžaduje komunikaci s jinou databází, v tom případě zmíněná část musí obsahovat svůj vlastní řetězec parametrů pro připojení k databázi.

Mimo běžné funkce systém obsahuje komunikační služby a rozhraní, které slouží z získávání dat od vozidel. Mezi tyto data, mimo *GPS* pozice vozidla, dále patří například současná rychlost, stav paliva, aktivita motoru, otáčky, teplota motoru a podobně. Tyto data jsou všeobecně zpracovávány přes *WCF* službu, která tyto data automaticky mapuje a ukládá na databázi. Následující schéma slouží jako abstraktní nákres struktury systému GX-GO.



Obrázek 2: Celkový pohled

Na obrázku 2 reprezentuje zjednodušenou představu toho, co vše systém GX-GO obsahuje. Mějme ale na paměti fakt, že nákres představuje jen zlomek reálné velikosti systému, slouží spíše jako abstraktní pomůcka pro představu jednotlivých částí. Jak je vidět, vozidla registrovaná v systému mají schopnost zasílat data o svém stavu, jako například současná *GPS* pozice, rychlost, otáčky, stav nádrží a podobně, a tyto data zasílá na naše všeobecné rozhraní, které umí tyto data zpracovávat a zapisovat to databáze tak, že ke všem typům dat potom existuje jednotný způsob přístupu, tedy odpadá nutnost řešit rozdíly vstupů u různých výrobců vozidel.

Samotný systém se potom dá rozdělit na databáze, moduly, pomocné knihovny, *WCF* služby a Webové aplikace (a její součásti). Databází je v systému hned několik, nejdůležitější je centrální databáze, kde jsou zpracovávány a uchovávány všechny aktuální i historická data jak klientů, tak jejich vozidel. Záložní databáze slouží k periodickému ukládání stavu hlavní databáze, z důvodu například technického selhání, aby pak bylo možné hlavní databázi snadno obnovit. V testovací databázi si vývojáři mohou testovat svá řešení úkolů a nemusí mít strach, že případná chyba se negativně odrazí na chodu celého systému.

Moduly, jako například Watchdog, Dispatcher nebo Routing, slouží ke zpracování k operací právě k tomuto modulu, a to jak kontrolu nad tvorbou nových dat, tak jejich získávání nebo úpravu. Pomocné knihovny v rámci systému obsahují nejčastěji sdílené modely pro data, všeobecné funkce nebo stavové číselníky. *WCF* aplikace zajišťují komunikaci separovanými částmi systému přes jednotné rozhraní. Služby slouží ke specifickým účelům aplikace, jako je například reverzní geo-kódování. A nakonec samotná webová aplikace a její součásti slouží k zobrazení informací pro uživatele. Mezi tyto informace patří prakticky všechna data, ke kterým má uživatel přístup.

Jak je z uvedených informací výše zřejmé, systém je rozsáhlý a neustále roste. V analýze požadavků pro modul proto bude muset být zvážen tento fakt tak, aby řešení modulu pro automatizované testování bylo permanentní, a nemuselo se měnit co několik měsíců, když se náhodou pozmění struktura systému.

3.2 Analýza požadavků pro modul automatizovaného testování

Modul pro automatizované testování má ve své finální fázi sloužit k testování všech změn v systému, které nastaly od posledního testu. V době svého běhu bude modul testování pracovat s celým systémem a všemi změněnými komponentami. To znamená, že tento modul bude požadovat schopnost detekovat chyby v řešení změny provedené programátory od doby posledního testu a tyto změny bude muset být schopen zavést do bezpečného prostření, kde bude možno je testovat. Prvním cílovým požadavkem je tedy schopnost reagovat na změny, tedy aby modul nezůstal statický.

K účelu získání nejaktuálnějších dat bude pravděpodobně nutné využít náš *SVN* server, kde se vždycky uchovává nejnovější verze systému. Proto modul bude potřebovat schopnost umět se serverem komunikovat. Kromě toho, server ve svém nastavení neuchovává projekty v jejich přeložené formě. Nicméně pro provedení testu potřebujeme projekty přeložit, to je tedy další věc, kterou bude muset být schopen modul provádět. K tomu účelu existuje knihovna v .NET, která dokáže přímo manipulovat s celými projekty.

Pokud se povede projekty korektně přeložit, bude nutné všechny složky s přeloženými soubory přesunout do separátního adresáře, který bude použit jako zdroj pro testování. Za doby běhu bude dobré ukládat relativní cesty k přeloženým projektům. Po úspěšném přeložení a přesunech, přikročíme k samotnému testování. V těchto testech bude nutné sestavit reálné cesty k .dll souborům cílových projektů a tyto cesty budou sestaveny právě na základě relativních cest

z překládání projektů. Pak už jen zbývá nechat všechny testy proběhnout, přečíst výsledky a tyto výsledky zavést do finální zprávy o testu. Tato zpráva bude sepsána ve formátu *XML*, pro efektivní čtení a snadné rozšíření.

Shrnutí analýzy testovacího modulu je tedy seznam akcí, které bude třeba zakomponovat do návrhu a ve výsledku i do analýzy. Mezi tyto akce patří:

- Vlastní vlákno pro modul
- Kontrola konfigurace spuštění
- SVN checkout
- SVN update
- Kontrola verzí
- Překlad projektů za běhu
- Kontrola chyb v překladu
- Identifikace testů
- Rozlišování typů testů
- Čtení výsledků testů
- Definice všeobecné XML struktury
- Zápis všech testů do reportu
- Kontrola chyb v celém modulu

3.3 Analýza technických požadavků

Pro korektní fungování modulu bude potřeba využít několik specifických technologií, protože se nejedná pouze o běžné testy. Proto bude nutné projít požadavky na jednotlivé typy testů a podle těchto požadavků najít potřebné technologické řešení.

Běžné testování, tedy testy nad metodami, které kontrolují výstupní hodnoty na základě určeného vstupu, jsou ty nejjednodušší. Pro jejich realizaci bude využita knihovna Unit Test. Tato knihovna nabízí plno možností a bude sloužit jako základ pro všechny testy. Unit Test funguje tak, že programátor určí vstupný hodnoty pro cílovou metodu, za předpokladu že cílová metoda požaduje vstup, na základě tohoto vstupu je vrácen výstup podle implementace metody. S výstupem se dá dál pracovat, například porovnat, jestli obdržený výstup se rovná očekávanému výstupu. Samozřejmě je možné provádět kontrolu několika způsoby, použití *Assert* je jen jednou z nich.

Databázové testy jsou o něco komplikovanější. Požadavky na modul specifikují, že bude nutné testy rozdělit na dva typy. Prvním typem jsou přímo testy návratových hodnot databáze. To je poměrně přímočaré řešení, s využitím Entity Framework stačí zavolat cílovou metodu, uložit si výsledné hodnoty a tyto hodnoty validovat. Druhý typ testu je mnohem složitější, jde o takzvaný *Mocking*. Mocking prakticky znamená, že cílená metoda je nahrazena novou mock metodou, která nám za všech okolností vrátí vždy výstup, jaký požadujeme. Tím pádem není testováno to, co metoda má vrátit, ale fakt, jestli se k cílové metodě vůbec dostaneme. Mock testy jsou velice efektivní pro testování větších metod, které ke své funkcionalitě volají několik dalších metod a protože mock dokáže nahrazovat výstup metod, hodí se nám právě v případě, kdy třeba nějaká metoda je nedostupná, nebo ve vývoji, nebo se často mění a programátora zajímá více jestli metoda funguje jako celek. Pro mock testy proto bude využita knihovna *MOQ*, která je speciálně vytvořena na tento typ testování.

Poslední typ jsou testy uživatelského rozhraní. Jinak řečeno, jde o způsob testování, který simuluje, jak se může uživatel v systému chovat. Jako chování v systému je možné uvést třeba modelový příklad, že uživatel se přihlásí do systému, klikne si na svůj učet, přejde na nastavení a změní své uživatelské heslo. Toto je například jeden scénář, který můžeme pomocí těchto testů simulovat. Pro tento test bude využit Selenium Web driver. Je tu ale jeden háček. Selenium pro svoji funkci vyžaduje webový prohlížeč, ale tuto možnost na aplikačním serveru nemáme. Z toho důvodu k Seleniu bude nutné přidat *PhantomJS*. Phantom je dodatek k Seleniu a funguje na principu virtuálního prohlížeče. Jinak řečeno, web se načte do paměti a s jeho strukturou se pracuje přímo v ovladači *PhantomJS*, ne na straně prohlížeče. S využitím *PhantomJS* tak odpadne nutnost používat prohlížeč a testy uživatelského rozhraní mohou bez problému probíhat na straně serveru.

Technické požadavky na modul se tedy dají shrnout jako seznam knihoven a rozšíření, které bude nutné do modulu připojit. Zejména jde o tyto knihovny a rozšíření:

- SharpSVN
- UnitTest
- NUnit
- Selenium.WebDriver
- PhantomJS
- Process / ProcessStartInfo
- Thread
- Moq

4 Seznámení s technologiemi

Analýza požadavků dala jasný náhled na to, jaké rozšíření bude třeba použít v modulu automatizovaného testování a v samotných testech v systému. Cílové technologie a knihovny, které je třeba rozebrat, jsou:

- SharpSVN
- Unit Test
- Moq
- Selenium.WebDriver
- PhantomJS

V této části budou výše uvedené technologie a knihovny probrány podle toho, jak budou v modulu Automatizovaného testování použity. Některé příklady již byly uvedeny v předchozí části, v této části budou rozebrány detailněji.

4.1 SharpSVN

SharpSVN, viz [8], je knihovna pro technologie .NET, která dokáže pracovat s *SVN* servery. Jejich funkcí a možností je mnoho, probrány budou pouze ty, které jsou pro modul relevantní. Pro následující ukázkové kódy nebude řešeno nic jako přihlášení klienta a konfigurace služby, tyto příklady mají za úkol dát přibližný pohled na práci s metodami knihovny SharpSVN.

Jako první z analýzy jasně vyplývá, že budeme potřebovat stáhnout celý repositář v případě, že na serveru ještě neexistuje, k tomu slouží metoda CheckOut, pro kterou existuje poměrně jednoduchý kód:

```
using (SvnClient client = new SvnClient())
{
  // CheckOut trunk into specified folder
  client.CheckOut(
    new Uri("http://demo/svn/trunk/"),
    "c://sharpsvn//CheckOutFolder"
  );
}
```

Výpis 1: SVN demo checkout

Další pro modul důležitou metodou je metoda Update. Tato metoda slouží synchronizaci verze na serveru s verzí lokálního repositáře. Jinak řečeno, všechny změny, které byly provedeny ve změnách na serveru, se odrazí v lokálním repositáři. Ukázkový kód funkcionality této metody je:

```
using (SvnClient client = new SvnClient())
{
  // Update the specified working copy path to the head revision
  client.Update("c://sharpsvn//CheckOutFolder");
}
```

Výpis 2: SVN demo update

Poslední důležitá metoda je metoda Commit. Jde o metodu, která pro změnu data na server nahrává, tedy pokud v modulu provedeme změnu, která se má odrazit na serveru, metoda Commit je k tomuto účelu ideální. Její použití je také poměrně snadné:

```
using (SvnClient client = new SvnClient())
{
  // Commit the changes with the specified log message
  SvnCommitArgs ca = new SvnCommitArgs();
  ca.LogMessage = "Demo commit message";
  client.Commit("c://sharpsvn//CheckOutFolder", ca);
}
```

Výpis 3: SVN demo commit

Knihovna *SharpSVN* všeobecně poskytuje metody a nástroje pro komunikaci a manipulaci s daty a *SVN* serverech. Z mnoha metod, které jsou v této knihovně poskytovány, již některé byly zmíněny. Existuje pak ještě několik dalších metod, které budou zapotřebí ke korektnímu splnění požadavků. Mezi tyto metody patří například ForceCredentials, CheckOut, Update, Commit, RevisionRemote, RevisionLocal, ConnectionStatus, ConnectionOpen a ConnectionClose.

4.2 Unit test

Další z vysoce užitečných knihoven pro .NET je Unit Test, viz [9]. Tato knihovna poskytuje celou škálu metod a konfigurací, které je možné použít pro testování implementovaných metod. Metody je možné testovat jak separátně a postupně, nebo podle předem definovaného scénáře.

Pro testové třídy je možné definovat tak zvané Initialize metody, tedy metody, které jsou spouštěny právě v moment inicializace. Existují tři druhy metod označených tímto atributem, metody, které se provedou jednou před tím než se začnou provádět všechny testy ve všech třídách, metody, které se provedou právě jednou pro celou testovací třídu a metody, které se provedou vždy před každým testem. Použití mají obě tyto metody, záleží spíše na programátorovi, který testy programuje, jestli je využije, nebo pro ně použití nemá. Příklad implementace těchto metod je vidět ve výpise 4:

```
[TestClass()]
public class ClassTest
{
 [AssemblyInitialize()]
 public static void AssemblyInit(TestContext context)
 {
  //here specify what should happen before all the tests start
  MessageBox.Show("Assembly Init");
 }
 [ClassInitialize()]
 public static void ClassInit(TestContext context)
 {
  //here specify what happens when class is initialized
  MessageBox.Show("ClassInit");
 }
 [TestInitialize()]
 public void Initialize()
 {
  //here specify what happens before each test
  MessageBox.Show("TestMethodInit");
 }
}
```

Výpis 4: Unit test Initialize demo

Opačné metody k Inicialize jsou metody s atributem Cleanup. Jak je zřejmé, tyto metody se provádějí po ukončení určité akce. Také zde jsou metody tří typů, metody, které se provádějí po dokončení všech testů v celém projektu, metody, které se provedou po ukončení všech testů v určité třídě a nakonec metody, které se provedou po dokončení každé testovací metody v určené třídě. Ukázkové kódy jsou vidět ve výpise 5:

```
[TestClass()]
public class ClassTest
{
 [AssemblyCleanup()]
 public static void AssemblyCleanup()
 {
  //here specify what should happen after all the tests are finished
  MessageBox.Show("AssemblyCleanup");
 }
```

```
 [ClassCleanup()]
 public static void ClassCleanup()
 {
  //here specify what happens when after all tests in a class
  MessageBox.Show("ClassCleanup");
 }
 [TestCleanup()]
 public void Cleanup()
 {
  //here specify what happens after each test
  MessageBox.Show("TestMethodCleanup");
 }
}
```

Výpis 5: Unit test Cleanup demo

Samotné testovací metody už jsou psány podle potřeb, co ja jak má být testováno. V testu je možné specifikovat vstupy, parametry nebo očekávané výstupy. Záleží na potřebách toho, co je od metody očekáváno. Samotná testovací metoda musí být označena atributem TestMethod, který jasně určuje, že se jedná o testovací metodu. Tato metoda by také měla obsahovat nějakou formu ověření, že bylo dosaženo požadovaného výsledku, nebo naopak, že došlo k selhání. Příklad takové metody je možné vidět ve výpise 6:

```
[TestClass()]
public class ClassTest
{
 [TestMethod()]
 public static void TestMethodDemo()
 {
  DemoTestClass target = new DemoTestClass();
  string expected = string.Empty;
  string actual= target.TryDemoMethod();
  Assert.AreEqual(expected, actual);
 }
}
```

Výpis 6: Unit test method demo

Unit testy jsou základ pro kontrolu testovacích scénářů. Všeobecně jsou jednoduché na tvorbu a snadné na použití. Principielně testují nejmenší jednotky v rámci systému, je ale možné je použít i na komplexnější testy. Pro testy bude proto potřeba používat korektní atributy a metody,

kde mezi tyto atributy a metody patří například TestClass, TestMethod, TestInitialize, ClassInitialize, TestCleanup, ClassCleanup, Assert a Eval.

4.3 MOQ

Knihovna *MOQ*, viz [10], je speciálně vytvořena pro testování podle *mock* principu. Mocking je typ testování, kde u komplexních metod, které volají ve svém těle několik metod, je možné nahradit výstup určité metody takovým výstupem, jaký je pro test požadovaný. Mock testování je velice užitečné ve chvílích, kdy je cílem testovat metodu jako takovou, jestli je schopna proběhnout kompletně sama o sobě a návratové hodnoty jsou druhořadá starost. Proto se mocking využívá v metodách, které volají metody, které zatím například nebyly implementovány, nebo byly implementovány, ale dochází v nich k nějaké opakované chybě, která by narušila testování.

V mock testování je do třídy Mock vložen objekt nebo třída, nad nimiž chceme testování provádět. Pomocí nastavení je možné definovat, která metoda má být mockována a jaká má být její návratová hodnota. Příklad jednoduché mock metody je možné vidět ve výpise 7.

Jak je v příkladu vidět, do Moq.Mock třídy je vložena třída Test, která je cíl pro toto demo testování. V Mock třídě je implementována metoda PerformAction nad třídou Test, která přijímá tři parametry. V části nastavení je proto definováno, že pokud bude přijat jakákoliv hodnota typu String, metoda se provede a její návratová hodnota bude vždy true.

Pak je třeba inicializovat samotnou třídu Test a do této třídy vložit mock objekt, který už je nastaven na správnou práci s cílovou metodou. Jediné co pak zbývá je metodu reálně zavolat a zkontrolovat, že opravdu proběhla. Tato metoda testování je použitelná pouze v případě, že třída přijímá v konstruktoru nadřazenou instanci jako parametr. Pro modul Automatizovaného testování bude zapotřebí implementaci upravit.

```
[TestClass]
public class MockTestClass
{
 [TestMethod]
 public void MockTestDemo()
 {
  //Create a mock object of a MailClient class which implements Test
  var mockTest = new Moq.Mock<Test>();
  //Mock the properties of mockTest
  mockTest.SetupProperty(client => client.TestValue1, "demoString").
     SetupProperty(client => client.Value, "12345");
  //Configure dummy method so that it return true
  mockTest.Setup(
   client => client.PerformAction( It.IsAny<string>(), It.IsAny<string>(), It.
       IsAny<string>()) ).Returns(true);
```

```
  Test test = new TestModel()
  {
   Val1 = "demo1",
   Val2="demo2",
   Val3="demo3"
  };
  //Use the mock object of Test instead of actual object
  var result = test.PerformAction(mockTest.Object);
  //Verify that the methods gets called exactly once
  mockTest.Verify(
   client => client.PerformAction(
    It.IsAny<string>(),
    It.IsAny<string>(),
    It.IsAny<string>()
  ), Times.Once);
 }
}
```

Výpis 7: Mock testing demo

4.4 Selenium web driver

Selenium, viz [11], je knihovna pro simulaci akcí uživatele přes webové rozhraní. Prakticky je v testovací metodě definováno, co se má stát na straně webové aplikace. Akcí uživatele se rozumí specifické chování, co může uživatel v systému dělat. Knihovna Selenium pracuje spolu s prohlížeči, je schopna se připojit ke každému typu prohlížeče, programátor tak může sledovat, jak probíhá test rozhraní. Ukázka takového testu je vidět ve výpise 8.

Samotný test je poměrně snadný. Na začátku je nutné inicializovat WebDriver, v tomto případě ten, který komunikuje s prohlížečem Google Chrome. Poté se prohlížeč přesune na určenou adresu. V hlavní metodě už jsou potom definovány kroky, které se mají provést, jako například hledání elementu, vložení hodnoty do textového pole, klik na tlačítko a podobně. Ve finální fázi je nutné otestovat, že bylo dosaženo stejného výsledku, jaký je očekáván.

Selenium poskytuje mnoho metod, pomocí kterých se dá manipulovat s uživatelským rozhraním. Některé tyto metody byly již uvedeny, v následujícím seznamu budou uvedeny ještě některé dodatečné. Mezi důležité metody a atributy, které bude třeba používat, patří například Navigate, GoToUrl, Perform, Action, Wait, MoveTo, FindElement, CreateElement, Insert a Write.

```
[TestClass]
public class SeleniumTestClass
{
 IWebDriver driver;
 string url = "http://demowebsite.com/demo/";

 [TestInitialize]
 public void TestSetup()
 {
  //here connect to a browser
  driver = new ChromeDriver();
  driver.Navigate().GoToUrl(url);
 }

 [TestMethod]
 public void SeleniumWebTestMethod()
 {
  driver.FindElement(By.PartialLinkText("Demo")).Click();
  var sku = driver.FindElement(By.CssSelector("#someTest")).Text;
  Assert.AreEqual("10001", sku);
  driver.FindElement(By.CssSelector("#anotherTest")).Click();
  //Verify if item has changed
  driver.FindElement(By.XPath(somePath));

  //Clikc Go to cart\
  driver.FindElement(By.CssSelector("#test2 ")).Click();
  var dd = driver.FindElement(By.XPath(anotherPath)).Text;

  //Verify If one phone given the cart.
  Assert.AreEqual("TestResult1", driver.FindElement(By.XPath(somePath)).Text);
  Assert.AreEqual("TestResult2", driver.FindElement(By.XPath(anotherPath)).Text
      );
  Assert.AreEqual("TestResult3", driver.FindElement(By.CssSelector("#someTest")
      ).Text);
 }
}
```

Výpis 8: Selenium demo - celé

4.5 PhantomJS

Jako poslední je třeba zmínit PhantomJS, který funguje jako nástavba nad běžným Seleniem. Tato nástavba umožňuje provádět simulace chování uživatele bez nutnosti inicializace prohlížeče. Tedy pokud jsou testy prováděny na straně serveru, není k nim zapotřebí prohlížeč. Samotný PhantomJS má velice snadnou inicializaci a ve své podstatě se od Selenia příliš neodlišuje. Pokud je tedy požadavek použít tuto nástavbu, k jejímu použití nám stačí podobný kód jako v demo příkladu:

```
 [TestClass]
public class PhantumJSTestClass
{
 [TestInitialize]
 public void TestSetup()
 {
  //Creating the instance of the PhantomJS driver
  phantom = new PhantomJSDriver();
  //Setting the default timeout to 30 seconds
  phantom.Manage().Timeouts().ImplicitlyWait(new TimeSpan(0, 0, 30));
  //Navigate to web
  phantom.Navigate().GoToUrl("https://www.demoweb.com");

  //then perform action to test
 }
}
```

Výpis 9: PhantomJS demo

Protože PhantomJS je nástavba nad Selenium, jeho metody jsou prakticky stejné, jako v čistém Seleniu. Nicméně PhantomJS není zatím ani zdaleka dokonalý a má jisté nedostatky, stále je to ale asi nejsilnější nástroj pro headless testing. Zmíněné nedostatky je třeba ladit individuálně, protože ne všechny akce z plného Selenia jsou stejně zpracovány ve Phantomu.

5 Návrh a Implementace

Část návrhu a implementace bude popisovat, jak byl modul Automatizovaného testování navržen na základě požadavků a kritéria, na základě kterých byla realizována samotná implementace. Kvůli rozsáhlosti systému je modul testování autonomní. Zbytek testů proto musí být viditelný, nemohou být skryté nebo přesunuté na jiný server.

Schéma systému je ukázané v sekci přílohy A na straně 69. Na zmíněném schématu je vidět detailnější pohled na systém a jeho komponenty. Podle barev, bílá jsou všechny komponenty, které v systému momentálně pracují. Modrá jsou navržené testovací projekty právě pro ty moduly, kde má testování nějaký smysl. Zelená je potom samotný testovací modul. Všechny testy jsou navržené podle toho, jakou komponentu mají testovat. Jak již bylo zmíněno v analýze, testů bude existovat několik typů. Na výše uvedeném schématu jednotlivé typy popsané nejsou, tyto testy budou detailněji popsané dále.

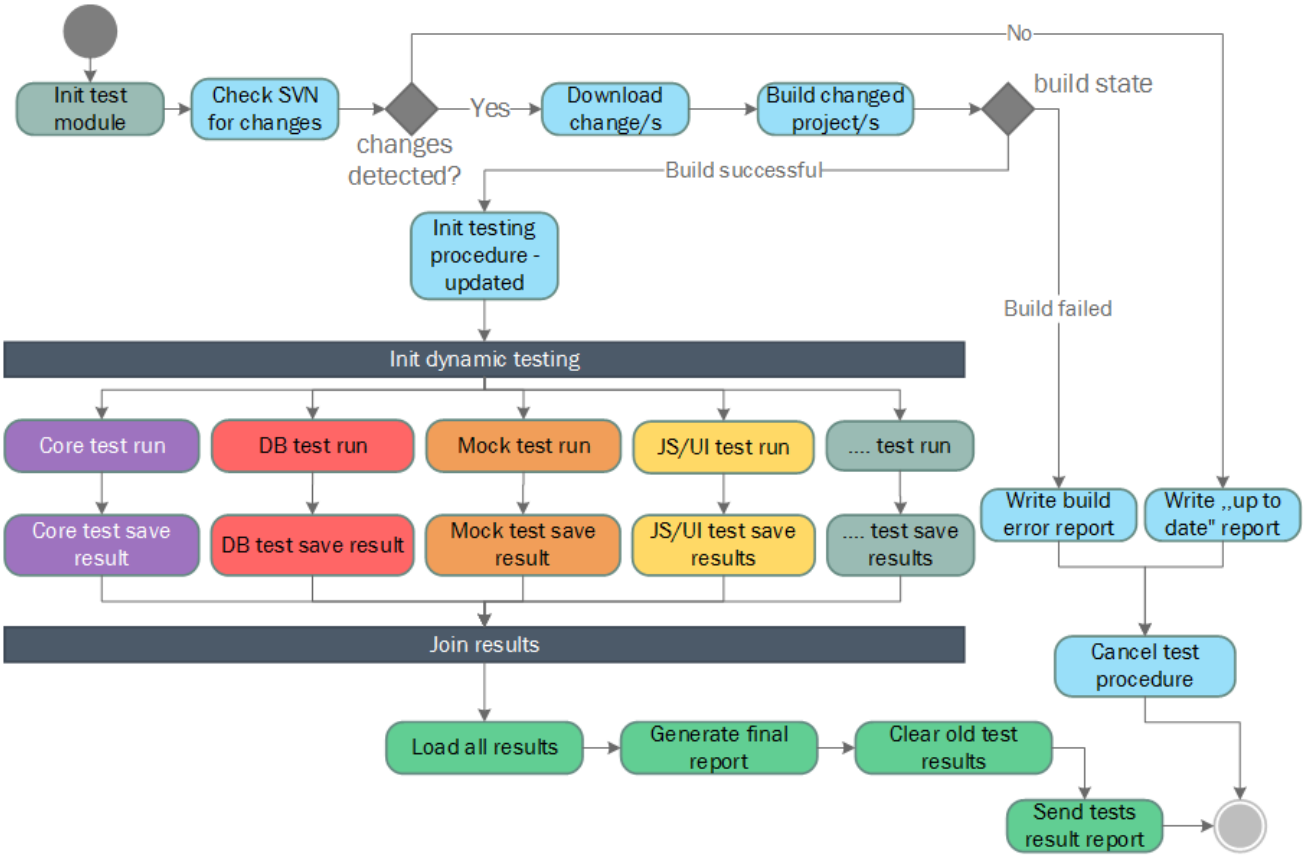
První část návrhu a implementace se proto bude zabývat celkově návrhem testovacího modulu, stejně tak jako jednotlivých stěžejních částí. Všechny tyto části budou pro lepší visuální představu popsány diagramy, aby čtenáři dosáhli lepšího porozumění toho, jak modul funguje. Všechny diagramy budou doprovázené textem jasně vysvětlujícím jednotlivé kroky, aby nedošlo k nedorozumění nebo chybnému vyložení popisovaného případu.

Druhá část návrhu a implementace bude popisovat návrh a tvorbu samotných testů. Jak je vidět na schématu výše, testů bude v systému mnoho, proto tato dokumentace je popíše pouze s pohledu jednotlivých tipů, kde ke každému přidá příklad. I zde budou testy doprovázeny diagramy podrobnými popisy, v tomto případě budou však diagramy menší.

5.1 Návrh a implementace modulu Automatizovaného testování

Jak již bylo řečeno, modul pro Automatizované testování bude fungovat jako autonomní komponenta systému. Návrh počítá s faktem, že celý proces poběží v samostatném vlákně. Proto na startu modulu se volá třída, která kontroluje vlákna a vytvoří nové vlákno právě pro tento modul. Po vytvoření vlákna se vytvoří instance pro třídu Scheduler, jejíž úkolem je kontrolovat, kdy se má test spouštět, na základě konfigurace modulu. Scheduler tedy v designovaný čas provede inicializaci části modulu zodpovědné za testování a provede spuštění. Část testování má několik fází, kde každá má specifické operace a parametry.

Diagram na obrázku 3 ukazuje celkový pohled na to, co je nutné provést v rámci jednoho běhu testů. Na začátku je tedy inicializována třída pro testování. První krok testů je ověření, jestli od posledního testu došlo na *SVN* k nějakým změnám. Tento krok je důležitý a to z toho důvodu, že pokud nedošlo ke změnám, tak by byly nakonec vráceny stejné výsledky jako při posledním testu, tím pádem tento scénář byl označen jako zbytečný a vyřazen.



Obrázek 3: Jedna iterace testu

První fáze je zkontrolování, jestli modul má na serveru již vytvořenou složku, kde si drží aktuální verzi *SVN* repositáře. Jeslti byla složka nalezena, pokračuje se na update, v opačném případě je nutné složku vytvořit a stáhnout celý aktuální repositář z *SVN* na server. Jakmile je update nebo download dokončen, zkontroluje se stav. Pokud nebyla objevena žádná chyba při komunikaci s *SVN*, proces splnil podmínku pro pokračování. V případě chyby je nutné proces ukončit, zapsat chybové hlášení a terminovat celý test.

Další krok procesu je build všech projektů. Metoda zodpovědná za build celého projektu v první fázi získá všechny projekty a zapíše si jejich cesty. Dále se pomocí iterace na všechny cesty vytvoří ProccesStartInfo, což je třída knihovny System.Diagnostics. Tato třída přijímá mnoho parametrů, pro potřeby buildu jsou cílové hlavně parametry Filename, který určuje cílový soubor, WorkingDirectory, který určuje cestu k projektu a Arguments, který funguje jako konfigurace a pomocí této konfigurace je proces nastaven tak, že má provést build. Veškerý výstup z konzole byl mimo jiné přesměrován na pomocné proměnné.

Na konci každého build procesu jsou oba výstupy kontrolovány pro možné chyby. Konkrétně je nutné provést kontrolu, jestli chybový výstup neobsahuje nějaké hlášení o nalezení chyby. V případě že chyba není nalezena, build proces pokračuje, dokud existuje nějaký stále nepřeložený projekt. V každém kroku je navíc nutné provést vyjmutí přeložených .dll z každého

projektu do speciální složky, která bude později použita k testování. Nicméně, pokud je v jakémkoliv kroku build procesu je nalezena chyba, celý proces je nutné terminovat, zapsat hlášení o chybě a ukončit celé testování.

Po úspěšném dokončení přeložení projektů je další na řadě samotné testování. Z části překládání projektů byly uloženy částečné cesty k přeloženým .dll souborům. Na základě těchto cest je pro každý test vytvořena absolutní cesta za použití relativní cesty, složky s přeloženými projekty a názvu cílového projektu. K účelu vytvoření cesty je použita metoda Combine z knihovny System.IO.Path. Každý test je zpracováván za použití ProccesStartInfo, v tomto případě ale konfigurace specifikuje použití programu MSTest, který se v programu Visual Studiu běžně používá jako výchozí program pro provádění testování. Výstupy z procesu jsou opět přesměrovány na pomocné proměnné, a to hlavně z důvodu že v konzoli by moc nápomocné nebyly a hlavně v konzoli je už nepřečteme, ne pokud modul poběží na straně serveru.

Po dokončení testovacího procesu, tedy po tom, jakmile jsou všechny testovací metody vyhodnoceny, proběhne kontrola výstupů z konzole. Pokud nebyla nalezena chyba, vezme se normální výstup z konzole a ten je převeden na hodnoty modelu pro testy. Tento model je poté uložen a testování pokračuje. I v případě nalezení chyby je pro změnu vytvořen chybový report pro daný test, ale testování není ukončeno, normálně pokračuje dál. Jakmile jsou všechny testy zpracovány, přechází se na poslední fázi.

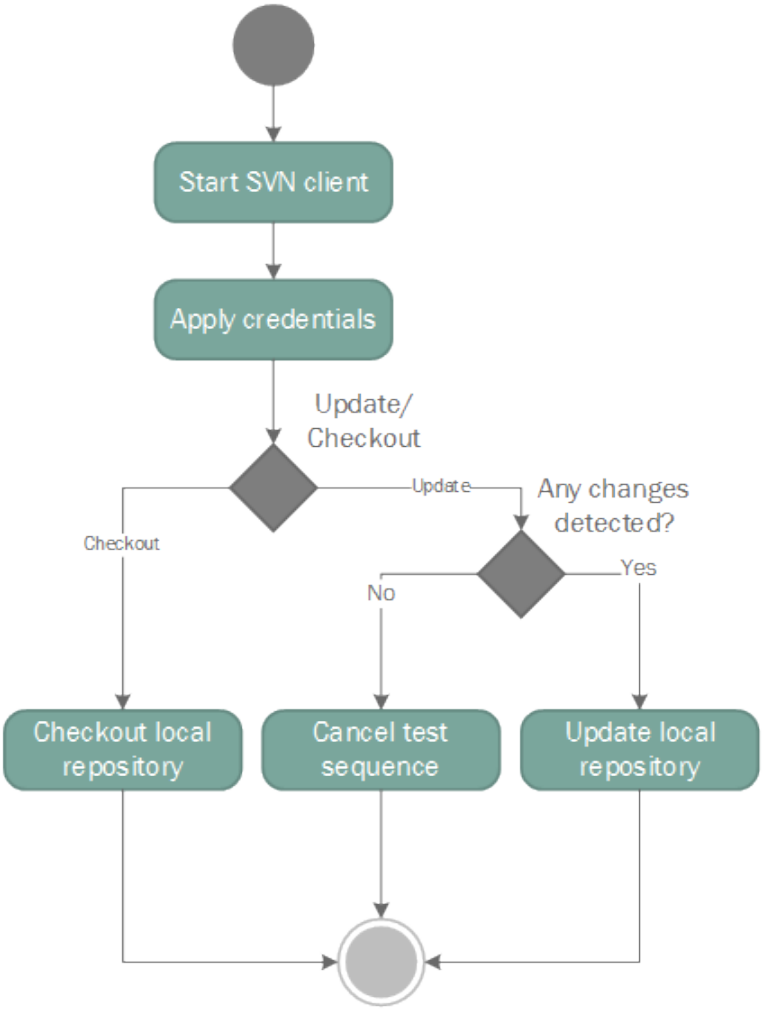
Poslední fází je vygenerování finální zprávy o stavu testů. Tato zpráva se generuje na základě hodnot v modelu, co kterého jsme průběžně ukládali hodnoty. Výsledná zpráva má formát *XML* dokumentu. Důvod pro tento formát je ten, že do budoucna se plánuje implementace rozhraní přímo v hlavním webovém systému, kde uživatelé s požadovanou úrovní registrace budou schopni kontrolovat výsledky proběhlých testů. Tento krok zatím nebyl plně zformulován, proto je zatím ve fázi přípravy. Po vytvoření dokumentu tedy testování končí a celý proces se vrací do Scheduleru, kde se bude čekat na další iteraci.

5.1.1 Návrh a Implementace SVN kontroly a zpracování projektů

Kontrola *SVN* a zpracování projektů je první stěžejní část modulu Automatizovaného testování. Jak již bylo naznačeno v předchozí části, tato část má několik fází a každá z těchto fází bude detailněji rozebrána níže. Pro lepší představu bude použit diagram.

Diagram na obrázku 4 je pouze zminimalizovaná varianta plné verze. Plná verze diagramu je uložena v příloze A na straně 70.

Prakticky než vůbec je možné začít testovat, musí být ověřeno, že současný čas odpovídá času specifikovaném v konfiguraci modulu. Jakmile je tedy tento čas potvrzen, třída pro testování *SVN* je inicializována a testování může začít. Pro jakoukoliv práci s *SVN* je nutné inicializovat *SVN* klienta a předat mu parametry pro přihlášení.



Obrázek 4: Kontrola SVN a zpracování projektů - minimalizovaná vezre

V tomto případě jsou parametry speciální účet vytvořený právě pro tento modul. Po předání parametrů přihlášení je specifikována konfigurace, pak už se stačí jen připojit. Nejprve je tedy provedena kontrola, jestli již existuje lokální repository. V případě, že je její existence potvrzena, proces může provést update, který je rychlejší a hlavně ve většině případů, menší. Ukázku toho, jak může takový update vypadat, demonstruje ukázkový kód ve výpise 10.

```
using (SvnClient svn = new SvnClient())
{
  svn.Authentication.ForceCredentials(USER, PSWD);
  svn.Authentication.SslServerTrustHandlers +=
  delegate(object sender, SvnSslServerTrustEventArgs e)
  {
    e.AcceptedFailures = e.Failures;
    e.Save = true;
```

```
  };
  svn.Update(path);
}
```

Výpis 10: SVN update - segment

Plná verze zdrojového kódu je dostupná v sekci přílohy B na straně 72. Jak znázorňuje ukázka zdrojového kódu, nejprve jsou nastaveny přihlašovací údaje klienta, v tomto případě přihlašovací údaje účtu pro modul testování. Poté se přidá nastavení ohledně práce s chybami. Dále se zkontroluje, jestli existují nějaké změny a pokud tyto změny byly nalezeny, proběhne update celé lokální repository. Pokud nedojde k žádné chybě, update je vyhodnocen jako úspěšný.

V případě, že lokální repositář ještě neexistuje, což se může stát ze dvou důvodů, prvním je, že toto je první test, nebo někdo lokální repositář smazal, je potřeba jej celkově stáhnout na server. K tomuto účelu slouží podobný kód, jehož ukázku je možné vidět ve výpise 11.

```
using (SvnClient svn = new SvnClient())
{
    svn.Authentication.ForceCredentials(USER, PSWD);
    svn.Authentication.SslServerTrustHandlers +=
    delegate(object sender, SvnSslServerTrustEventArgs e)
    {
     e.AcceptedFailures = e.Failures;
     e.Save = true;
    };
    svn.GetInfo(repos, out info);
    svn.CheckOut(repos, path);
}
```

Výpis 11: SVN checkout - segment

Plná verze zdrojového kódu je dostupná v sekci přílohy B na straně 73. Tento kód je velmi podobný, existuje tu ale několik rozdílů. Prvním je tvorba složky pro repositář hned na začátku. Pro přihlášení i chyby je nastavení shodné. Pak je nutné získat informace o *SVN* repositáři, které dál budou použity. Samotná metoda pro checkout přijímá dva parametry, a to adresu zdroje, a cestu k cíli. Celý proces stažení z *SVN* je časově náročný a zabere i několik minut, to ale závisí na kvalitě připojení. Jakmile je ověřeno, že stažení bylo úspěšně dokončeno, proces pokračuje dále.

Pak ještě může nastat případ, že na *SVN* nebyly nalezeny žádné změny od posledního testu. V tom případě můžeme celé testování zrušit, není důvod testovat znovu něco, co již testované bylo, jen aby jsme obdrželi stejné výsledky.

Další fází je přeložení všech projektů. Je nutné přeložit vše, ne jen některé, protože pokud by v nějakém nastala chyba například kvůli překlepu programátora, je nutné tuto informaci vědět.

Proces překladu je poměrně přímočarý. Nejprve se získají všechny soubory s koncovkou csproj, tak je zajištěno, že některý projekt nebude opomenut. Ukázkový kód pro překlad je uveden ve výpise 12.

```
ProcessStartInfo startInfo = new ProcessStartInfo()
{
 //Attribute definition here
};
using (Process exeProcess = Process.Start(startInfo))
{
 //Test the build ctream
 exeProcess.WaitForExit();
}
DirectoryCopy(pathBinDebug, pathWCF, true);
```

Výpis 12: Project build - segment

Plná verze zdrojového kódu je dostupná v sekci přílohy B na straně 74. Složka pro výsledné .dll soubory se vytváří před samotným začátkem překladu. Potom se provádí iterace pro všechny projekty, kde každý projekt je přeložen samostatně a také samostatně vyhodnocen. Opět je použit ProccesStartInfo, v tomto případě je nutné specifikovat prostředí a konfiguraci tak, aby pracovala s nástrojem MSBuild, který je také jednou z klíčových částí prostředí Visual Studio.

Překlad projektu funguje na stejném principu jako kdyby byl překlad prováděn přímo ve Visual studiu.

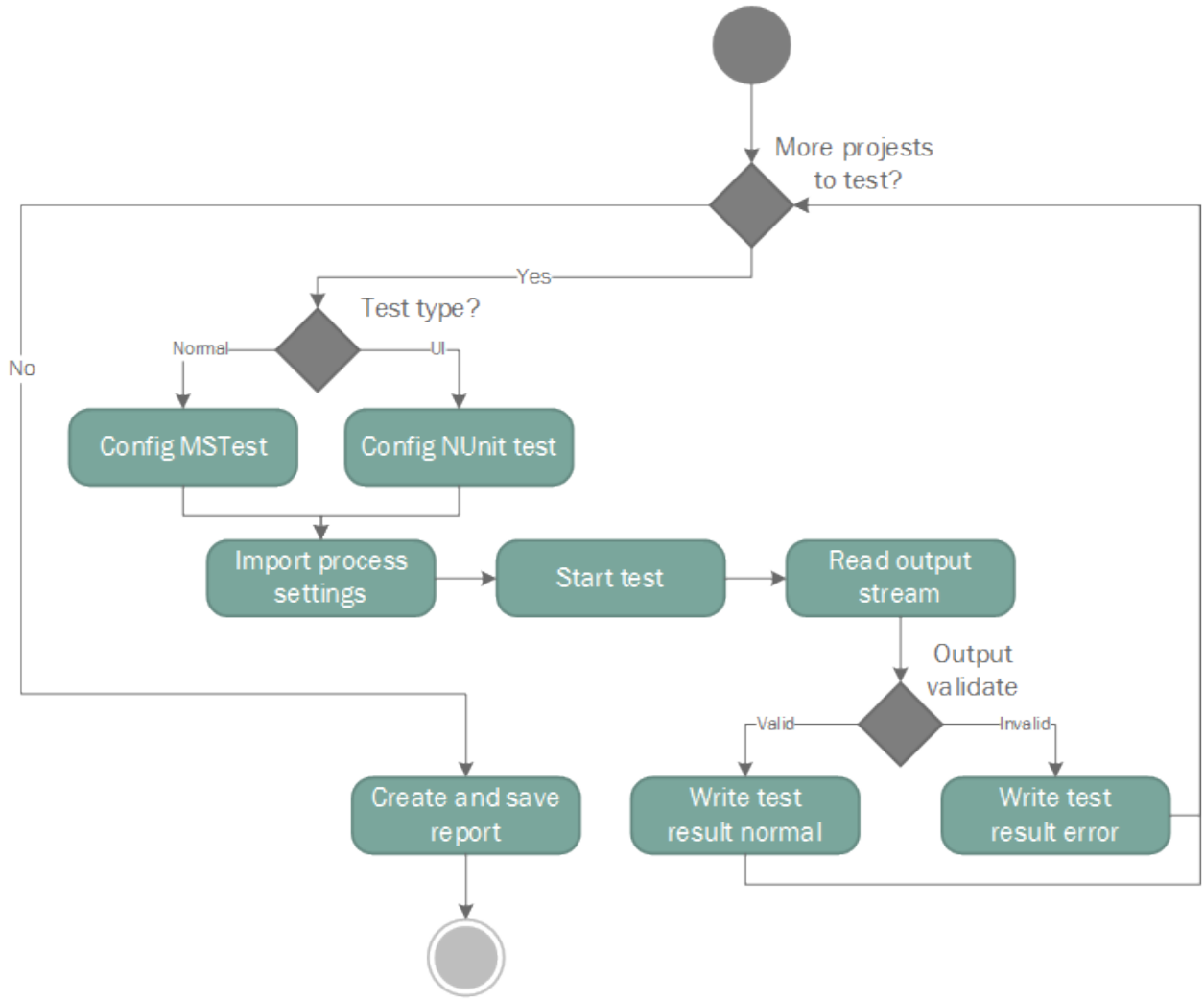
Za normálních okolností konzole by podávala informace o tom, jak překlad proběhl, v případě modulu automatizovaného testování je výstup opět přesměrován na pomocné proměnné, ze kterých můžeme za překladu číst příchozí informace. Pokud by se stalo, že některá z těchto informací má označení Error, proces překladu je nutné hned ukončit, zapsat záznam o chybě a tento záznam uložit. Pokud však překlad proběhne správně, je možné do složky pro testování vypublikovat přeložené .dll soubory pro daný projekt. Zároveň se také ukládá název projektu a jeho relativní cesta, kde obě tyto informace jsou později využity u samotného testování.

Překlad probíhá tak dlouho, dokud existuje ještě nějaký nepřeložený projekt v lokálním repositáři. Po provedení překladu a vypublikování .dll souborů je další na řadě je samotné testování. K procesu testů se je možné dostat pouze v případě, že stažení z *SVN* nenahlásilo chybu, překlad projektů proběhl v pořádku a všechny cílové projekty se povedlo vypublikovat do testovací složky.

5.1.2 Návrh a Implementace testování cílových projektů

Samotné testování je o něco jednodušší. Jak již bylo řečeno dříve, testování je možné provádět jen pokud se korektně provede *SVN* kontrola a překlad projektů. V testech proto se už pracuje

přímo s výsledky a tento proces je o něco méně náročný. Pro lepší vizuální představu je možné se podívat na diagram na obrázku 5.



Obrázek 5: Testování projektů - minimalizovaný

Celý diagram je dostupný v sekci přílohy A na straně 71. Běh testů probíhá také na několik fází. Nejprve je inicializována hlavní testovací třída a namapuje se absolutní cesta k nástroji MSTest. Protože by bylo zbytečné mapovat cestu pro každý test zvlášť, je efektivnější tuto akci provést na začátku pouze jednou a pak se jen dotazovat na výsledek. Dále jsou připraveny cesty pro samotné testy. Tyto cesty jsou sestaveny z relativní cesty k samotným .dll objektům a názvu cíleného projektu k testování. Na základě těchto parametrů je pro každý test sestavena absolutní cesta, která jednoznačně identifikuje objekt na diskovém úložišti.

Jakmile jsou cesty připravené, je zahájen iterační průchod všemi cestami a protože každá cesta vede přímo k testu, pro každou cestu je třeba vytvořit speciální test se specifickými parametry. Pro testování existuje testovací třída, která je inicializována pro každý test. V této

třídě SingleTest je metoda PerformTest, která přijímá dva parametry. První je identifikátor samotného projektu k testování, druhý je plná cesta k projektu, který je cíl tohoto testu.

V testu se nejprve inicializuje model, do kterého se po testu uloží výsledky tohoto testu. Jako další je vytvořena instance třídy Stopwatch, která má v testu za úkol měřit čas, jaký byl zapotřebí na zpracování celého testu. Poté aktivaci Stopwatch je třeba upravit cestu k .dll objektu pro projekt tak, aby byla akceptována konfigurací pro ProccesStartInfo. Samotná instance ProccesStartInfo přijímá několik speciálních parametrů, které definují, že cílená cesta bude využita pro testování. K tomu je dále třeba definovat cestu k nástroji MSTest.

Dále už jen stačí přesměrovat výstup na pomocné hodnoty mimo konzoli, aby bylo možné číst aktivitu procesu. Na úplném konci procesu jsou oba vstupy přečteny, aby se zjistilo, jestli nedošlo v testování k chybě. Jak v případě, chyby, tak v případě bezproblémového testu, je proveden zápis do výsledného modelu. Tímto okamžikem specifikovaný test končí, výsledkový model je uložen a pokračuje se k dalšímu testu. Ukázku testu je možné vidět ve výpise 13.

```
Process myProcess = new Process();
ProcessStartInfo myProcessStartInfo = new ProcessStartInfo()
{
  // Attributes for ProcessStartInfo
};
myProcess.StartInfo = myProcessStartInfo;
myProcess.Start();

string outputErrors = myProcess.StandardError.ReadToEnd();
string output = myProcess.StandardOutput.ReadToEnd();

myProcess.WaitForExit();
myProcess.Close();
ReportHelper.FinalizeTestModel(output, outputErrors, model);
```

Výpis 13: Project test - segment

Plná verze zdrojového kódu je dostupná v sekci přílohy B na straně 76. Po ukončení všech testů nastává poslední fáze a to vytvoření finální zprávy o testech z modelů, které byly zmíněné v předchozím odstavci. Vygenerování konečné zprávy probíhá přímo ze seznamu modelů, které byly výsledkem všech provedených testů. Zpráva bude ve formátu *XML* za pomocí třídy System.XML.XMLDocument, která umožňuje velmi snadnou tvorbu a manipulaci se soubory ve formátu *XML*. Výsledný dokument obsahuje jak všeobecné informace o jednotlivých testovaných projektech, tak informace ohledně jednotlivých testovaných metod a hlavně informaci o stavu testu pro danou metodu.

Tvorba probíhá postupně. Nejprve je nutné vytvořit samotný hlavní soubor. Pro tento soubor je tedy vytvořeno jméno, které je složené ze dvou částí, první, která identifikuje, že jde o výsledek

testu a druhá, která obsahuje časový identifikátor, který říká kdy byl test proveden. Dále je nutné získat objekt, který obsahuje všechny výsledky pro všechny testy. Tento objekt je získán z třídy DataHolder, která, kromě údajů o testech, obsahuje několik dalších důležitých informací, které je nutné sdílet v rámci celého modulu. Po získání objektu s výsledky je zavolána metoda pro vygenerování *XML* zprávy.

V metodě pro tvorbu finální zprávy se pomocí iterace projde kompletní seznam všech výsledků, které byly obdrženy z třídy DataHolder. Pro každý tento testovaný projekt se do *XML* dokumentu zapíší nejprve všeobecnější informace, jako například název projektu, kolik metod bylo testováno a jak dlouho test zabral. Po zapsání těchto informací se přejde na vnořenou iteraci, která projde všechny výsledky testovaných metod v rámci tohoto projektu. Výsledky metod obsahují převážně jen název metody a výsledek testu. Tato iterace tak projde všechny výsledky, dokud ještě existují nějaké výsledky, které nebyly zapsány. Po ukončení vnitřní iterace je *XML* struktura obsahující informace o projektu přidána do hlavního dokumentu a iterace pokračuje na další projekt, kde se celý proces opakuje. Příklad tvorby *XML* je možné vidět ve výpise 14.

```
XmlElement report = parentDoc.CreateElement(
String.Empty, "Report", String.Empty );
XmlElement name = parentDoc.CreateElement(null, "Project_name", null);
// Fill ProjectName
XmlElement state = parentDoc.CreateElement(String.Empty, "Test_state", String.
    Empty);
// Fill Test state
foreach(var method in data.TestCaseList)
{
 // Write test cases results
}
rep.AppendChild(report);
```

Výpis 14: XML document create - segment

Plná verze zdrojového kódu je dostupná v sekci přílohy B na straně 77. Po provedení tvorby dokumentu zbývá už jen dokument uložit. Dokument je uložen na jednotné místo specifikované v konfiguraci modulu. Jako ukázku toho, jak vypadá výsledná zpráva to testu se je možné podívat na přiložený *XML* dokument ve výpise 15.

```
<?xml version="1.0" encoding="UTF-8"?>
<Reports>
 <Report>
  <Project_name>DemoUnitTest</Project_name>
  <Test_state>2/2 test(s) Passed</Test_state>
  <Elapsed_time>00:00:01</Elapsed_time>
  <Method_results>
```

```
    <Method>
     <Name>DemoUnitTest.UnitTest1.TestMethod1</Name>
     <Result>Passed</Result>
    </Method>
    <Method>
     <Name>DemoUnitTest.UnitTest1.TestMethod2</Name>
     <Result>Passed</Result>
    </Method>
   </Method_results>
  </Report>
</Reports>
```

Výpis 15: XML result demo

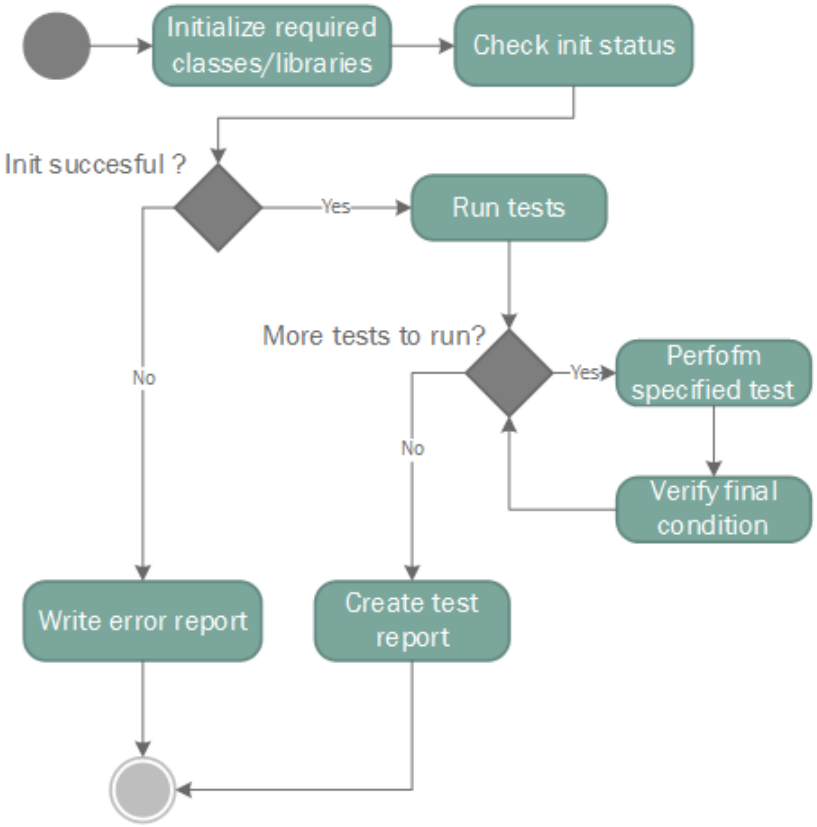
5.2 Návrh a Implementace testů pro cílové projektové komponenty

Následující část se bude zabývat návrhem a implementací samotných testů pro jednotlivé projekty, u kterých je vyžadováno testování. Jak již bylo naznačeno, samotné testy jsou rozděleny na několik typů, podle toho, který projekt mají testovat nebo jakou konkrétní funkcionalitu je nutné vyzkoušet. Všechny testy vycházejí z knihovny Microsoft.VisualStudio.TestTools.UnitTesting. Tato knihovna je použita jako základ pro všechny testy, avšak pro specifické typy je přidáno několik dalších knihoven pro dosažení požadované funkcionality.

Tato část se proto bude zabývat rozborem a konkrétními ukázkovými příklady pro jednotlivé typy testů, aby čtenáři dosáhli lepšího pochopení faktu, jak se testy dají rozdělovat a jak se jednotlivé rozdíly odrazí v implementaci těchto testů. Návrh a implementace proto bude rozdělena na čtyři části, podle toho, o jaký typ testů půjde. Cílené type testů jsou Core testy, tedy testy zaměřující se na funkcionalitu hlavně metod v jádru systému, Databázové testy, které kontrolují korektnost vrácených dat přímo z databáze, Mock testy, které testují spíše fakt, jestli cílené metody probíhají správně, než fakt jestli vrací správná data, a UI testy, které simulují aktivitu uživatele přímo na webovém rozhraní systému GX-GO.

5.2.1 Návrh a Implementace Core testů

Core testy, neboli testy metod v jádru systému, jsou prvním typem, který se zde bude rozebírat. Tyto testy se zaměřují primárně na testování metod v projektech, které pracují na pozadí systému. Principem tohoto testování je tedy odzkoušet, jestli různé metody požadované pro různé části systému pracují korektně a vracejí korektní výsledky vzhledem k vloženým parametrům. Samotný test je prakticky vždy jedna metoda cílená na reálnou metodu v testovaném projektu, plus vstupní data, kontrola výstupních dat a ověření, že nedošlo k chybě. Příklad toho, jak je možné provést tento test je uveden v jednoduchém diagramu na obrázku 6.



Obrázek 6: Core test demo

V první fázi testu se provádějí inicializační metody, které se spouštějí buď na začátku startu celého projektu, na začátku inicializace třídy pro testování nebo před začátkem provedení každé testovací metody. Pokud inicializační fáze proběhne v pořádku, přejde se k samotnému testování. V každém testu se testována jedna nebo více metod, pro které programátor specifikuje vstupní parametry, stejně tak jako parametry, které jsou očekávány na výstupu.

Výstupní parametry je nutné validovat, jestli odpovídají očekávanému výsledku. K tomu slouží metoda Assert, která je součástí knihovny UnitTesting. Tato metoda obsahuje mnoho variant, jak můžeme testovat shodu očekávaného výstupu proti reálnému výstupu. Jeden z příkladů, jak může test vypadat, je vidět ve výpisu 16.

```
[TestMethod]
public void GetReportDataMileageTest()
{
 List<int> ids = new List<int>();
 ids.Add(3);

 var result = ReportsCore.GetReportDataMileage(ids, new AppContext() { /*
    Context*/ } );
 Assert.IsTrue(result.Count != 0);
```

```
}
```

Výpis 16: Core test demo - segment

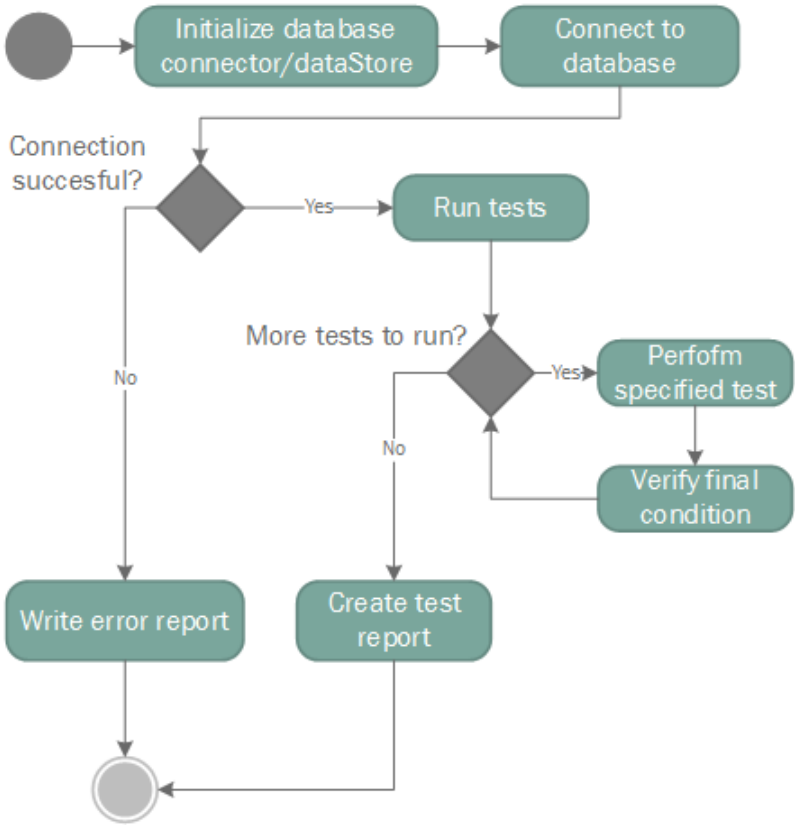
Plná verze zdrojového kódu je dostupná v sekci přílohy B na straně 79. V tomto ukázkovém testu, konkrétně v jeho plné verzi, je inicializační metoda s atributem ClassInitialize, který definuje, že metoda se provede právě jednou a to při initializaci této třídy. Uvnitř této metody se aktivuje třída AutoMapper, která má za úkol automaticky namapovávat databázové objekty na sdílené modely používané v systému přes sdílenou knihovnu Shared.DataContracts. Po inicializaci mapperu se automaticky provede první metoda v seznamu, v tomto případě metoda GetReportDataMileageTest.

Ve zmíněné metodě je inicializován seznam číselných hodnot, v tomto případě jde o seznam ID, které identifikují vozidla, pro které je požadován výsledek. Dále je volána samotná metoda GetReportDataMileage, která se nachází ve třídě ReportsCore. Kromě seznamu ID vozidel, metoda dále požaduje jako vstup objekt AppContext, který obsahuje dodatečné informace o uživateli, který po metodě požaduje data. Tento objekt slouží také jako kontrola pro neoprávněný požadavek na data, tedy pokud specifikovaný uživatel nemá pro získání dat dostatečná práva.

Proměnná result by po provedení metody GetReportDataMileage měla obsahovat seznam objektů Mileage, které byly získány z databáze na základě vstupních parametrů. Poslední fází testu je Assert, v tomto případě Assert.IsTrue, který kontroluje pravdivost tvrzení specifikovaného uvnitř. Pro tento případ jde o tvrzení, že vrácený seznam má délku, která je různá od hodnoty nula. Pokud tedy je tato podmínka splněna, test je vyhodnocen jako splněný a pokračuje se na další metodu k otestování.

5.2.2 Návrh a Implementace Databázových testů

Další na řadě jsou databázové testy. V těchto typech testů se kontrolují přímo výstupní databázové hodnoty na základě přiřazené vstupní hodnoty. Pro připojení je nutné definovat klienta pro připojení, který mimo jiné obsahuje connection string definující databázi, na kterou bude dotaz cílen. Databázové testy, stejně jako ostatní, pracují na principu specifikace vstupních parametrů pro metodu, inicializace cílené metody se specifikovanými parametry, získání návratové hodnoty a kontrola této hodnoty, aby bylo možné ověřit korektní chod testu. Pro představu databázového testu je přiložen diagram na obrázku 7.



Obrázek 7: Database test demo

Na diagramu je vidět, že nejprve se inicializuje samotné spojení s databází. Tento krok je nutný v případě, že je požadována komunikace s jinou databází než z tou výchozí. Pro princip testování se používá záložní databáze, která se aktualizuje každý den a funguje mimo jiné jako záloha pro hlavní databázi v případě, že by došlo ke ztrátě dat, nebo poškození struktury hlavní databáze. Po inicializaci pak testování pokračuje na samotné metody.

Testy probíhají na stejném principu jako zmíněné Core testy, rozdíl je ten, že u Databázových testů jsou kontrolovány přímo návratová data z databáze, bez nějaké dodatečné modifikace přes další metody. Stejně jako u ostatních testů je nutné specifikovat vstupní parametry pro test, tedy programátor specifikuje, jak budou vypadat vstupní data a pro tyto data ví, jaký má očekávat výsledek. Samotná kontrola funguje na stejném principu, tedy za použití metody Assert.

Jak je vidět na diagramu, testování probíhá tak dlouho, dokud ještě existují metody, které nebyly otestovány. Po ukončení testů se zapíší výsledky pro výslednou zprávu a specifikovaný test končí. Příklad toho, jak může takový test vypadat je vidět v ukázkovém kódu ve výpise 17.

```
[TestMethod]
public void GetDeviceProfileInputTest()
{
 var appContext = new DataContracts.AppContext() { /*Context*/ };
 var iut = new DeviceStore { AppContext = appContext };
```

```
 var result = iut.GetDeviceProfileInput(3);
 Assert.IsTrue(result != null);
}
```

Výpis 17: Database test demo - segment

Plná verze zdrojového kódu k tomuto testu je dostupná v sekci přílohy B na straně 80. V uvedeném ukázkovém kódu, v jeho plné verzi, jsou dvě testovací metody, metoda GetDeviceProfileInputTest a metoda CreateTest. Každá z těchto metod komunikuje s právě jednou další metodou v tak zvané Store struktuře. Obě metody mají implementované vstupní parametry, na základě kterých jsou získávány data z databáze.

Pro metodu GetDeviceProfileInputTest je vidět, že jako první je vytvořen AppContext, který byl již zmíněn dříve. Tento objekt opět obsahuje parametry uživatele, který požaduje z databáze výsledek. Po inicializaci tohoto AppContext je tento objekt použit jako vstupní parametr pro inicializaci třídy DeviceStore, která je použita ke komunikaci s databázovou tabulkou Device, případně dalšími tabulkami, které by mohly být ovlivněny změnami v tabulce Device.

Po inicializaci DeviceStore je zavolána metoda GetDeviceProfileInput, která přijímá číselný vstupní parametr, v tomto případě ID vozidla, ohledně kterého chce programátor zjisti dané informace. Výsledek této operace je objekt ProfileInput, který obsahuje informace o profilových vstupech pro vozidlo. Nakonec po získání tohoto objektu se přes metodu Assert ověří podmínka, že výsledný objekt obsahuje data, tedy že není prázdný.

```
[TestMethod]
public void CreateTest()
{
 var appContext = new DataContracts.AppContext() { /*Context*/ };
 var iut = new DeviceStore { AppContext = appContext };
 var result = iut.Create(new Repository.Device() { /* Device attributes */ },
     appContext.UserID);

 Assert.IsTrue(result != null);
}
```

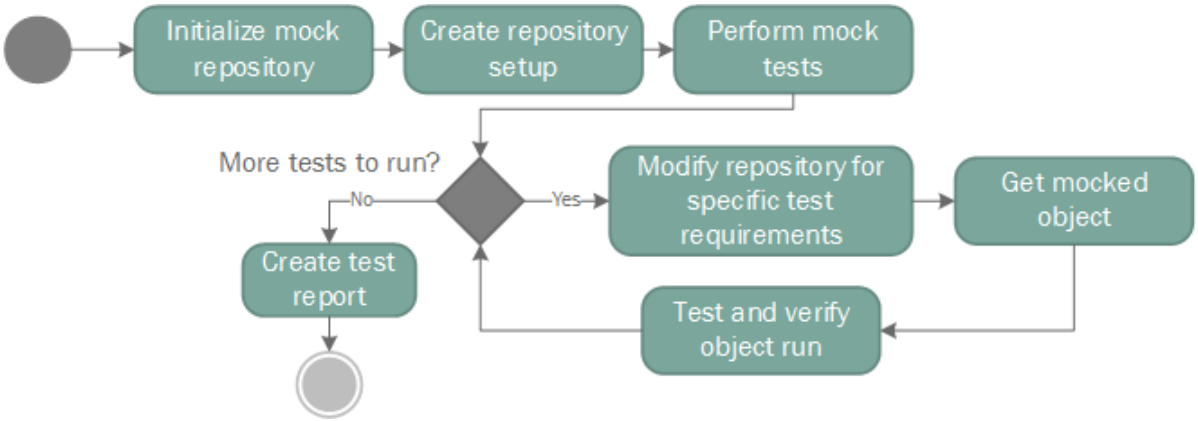
Výpis 18: Database test demo - segment

Druhá metoda, tedy CreateTest, testuje tvorbu nových dat pro tabulku Device. Stejně jako u předchozího popisu, je nutná specifikace AppContext. Dále se inicializuje DeviceStore s objektem AppContext jako vstupním parametrem. Cílová metoda pro tento test je metoda Create, která přijímá jako vstup objekt typu Device. Proto pro testování je specifikováno testovací vozidlo s parametry, které jsou později použity k jeho nalezení a vymazání.

Po testu metody Create je obvykle jako další v sekvenci test metody Update a nakonec test metody Delete, kde všechny tyto metody operují nad stejným vozidlem. Jak je také vidět, kromě objektu Device metoda přijímá další parametr, a to číselnou hodnotu UserID, která je v databázi použita jako označení toho, který uživatel změnu provedl. Získaná návratová hodnota je v tomto případě stejný databázový objekt jako ten, který byl vložen na vstupu, s tím rozdílem že databáze do něj přidá dodatečné atributy, které jsou generovány automaticky. Po získání tohoto objektu je opět ověřeno, že objekt není prázdný a test končí.

5.2.3 Návrh a Implementace Mock testů

Mock testy, jak již bylo zmíněno, jsou trochu jiný typ testů než jaký byl doposud popisován. Tento typ testů netestuje ani tak výsledná data proti očekávaným datům, ale spíše fakt, jestli metoda je schopna proběhnout dokonce. Mock testování je vhodné pro případy, kde například část funkcionality určité metody je nefunkční. Proto v tomto testu není problém nefunkční kód obejít konkrétní specifikací v nastavení a donutit ji tak, aby vrátila vždy požadovaný výsledek, nehledě na fakt, že za normálních okolností by vrátila chybu. Způsob, jak probíhají mock testy v modulu Automatizovaného testování je vidět na obrázku 8.



Obrázek 8: Mock test demo

V Mock testu je nejprve nutné inicializovat repositář, přes který probíhá komunikace s cílovými metodami. Mock objekt, který bude dále fungovat místo reálného repositáře, se v této fázi nastaví na výchozí nastavení, které se bude obnovovat pro každý test. Po této fázi jsou provedeny samotné mock testy. V každém testu se to výchozího nastavení musí dále specifikovat, kterou metodu a jak bude nutné obejít, aby byla opravdu testována jen funkcionalita a nedocházelo k žádné komunikaci s databází nebo jinými metodami.

Pro každý test je nutné specifikovat podmínky testu a hlavně správný override pro části kódu, které nejsou požadovány, aby prováděly svou pravou funkci, ale stačí, aby pouze vrátili nějakou definovanou hodnotu. Pro lepší představu je uveden ukázkový kód mock testu ve výpise 19.

```
[TestMethod]
public void ApiSourceStoreGet()
{
 mockDataStore.Setup( s => s.GetDataStore<IApiSourceStore>(
  It.IsAny<AppContext>()
 )).Returns<ApiSourceStore>(store =>
 {
  var mock = new Mock<IApiSourceStore>() { CallBase = true };
  mock.Object.AppContext = appContext;
  mock.Setup(s => s.Get(It.IsAny<int>())).Returns(new ApiSource());
  return mock.Object;
 });

 var mockObject = mockDataStore.Object.GetDataStore<IApiSourceStore>(null);
 var result = mockObject.Get(1);
 Assert.IsInstanceOfType(result, typeof(ApiSource));
}
```

Výpis 19: Mock test demo - segment

Plná verze zdrojového kódu k tomuto testu je dostupná v sekci přílohy B na straně 81. Ukázkový kód obsahuje metodu TestInit s atributem TestInitialize, který specifikuje, že tato metoda se provede vždy před začátkem každé testovací metody. Tato metoda je nyní součástí pouze plné verze kódu, protože její důležitost k celkovému pochopení je zanedbatelná. Vždy je nutné AppContext a Mock repositář pro každý test nastavit na výchozí hodnoty. Po nastavení proces pokračuje na testovací metodu ApiSourceStoreGet.

V metodě ApiSourceStoreGet se jako první provádí nastavení pro mockDataStore. Nastavení pro Mock se provádí přes parametr .Setup, kde se specifikuje metoda, která se má obejít, v tomto případě metoda GetDataStore, která za normálních okolností vrátí objekt ApiSourceStore. Tento objekt by normálně obsahoval parametry pro spojení s databází a práce s daty. Jak je vidět, metoda GetDataStore typu IApiSourceStore normálně přijímá vstupní parametr typu AppContext, který je v tomto případě nahrazen funkcí It.IsAny, která nahradí cílený objekt čímkoliv, co stále splňuje podmínku korektního zavolání metody.

Další v řadě je parametr .Return, který specifikuje návratovou hodnotu mock metody, která byla popsána v předchozím odstavci. V tomto případě je cílem dostat objekt typu ApiSourceStore, ale protože v nastavená stále nebyla obejita metoda, na kterou je test mířen, je nutné nastavení dále specifikovat. Jak je vidět dále, v dodatečném nastavení se vytváří nový mock objekt typu IApiSourceStore, kterému je dále přížazen objekt AppContext, který byl vytvořen v inicializační fázi.

Hlavní krok tohoto nastavení je sepcifikace pomocí .Setup, kde se definuje, že metoda Get, která bude přijímat jakoukoliv číselnou hodnotu, vždy vrátí nový objekt typu ApiSource. Po dokončení této specifikace je už jen nutné definovat, že mock objekt z vnořeného nastavení bude vrácen mockovanou metodou z hlavního nastavení.

Po úspěšném nastavení mock objektu je už jen nutné metodu otestovat, jestli opravdu probíhá tak, jak má. Proto je tato metoda dále zavolána s číselným parametrem na vstupu a proměnná result zde bude sloužit k obdržení návratové hodnoty. I v tomto případě je využita metoda Assert, konkrétně její funkce IsInstanceOf, která jasně prokáže, jestli výsledný vrácený objekt je opravdu toho typu, jaký je požadován. Po kontrole této podmínky je test ukončen a opět bude zavolána inicializace před začátkem dalšího testu v pořadí.

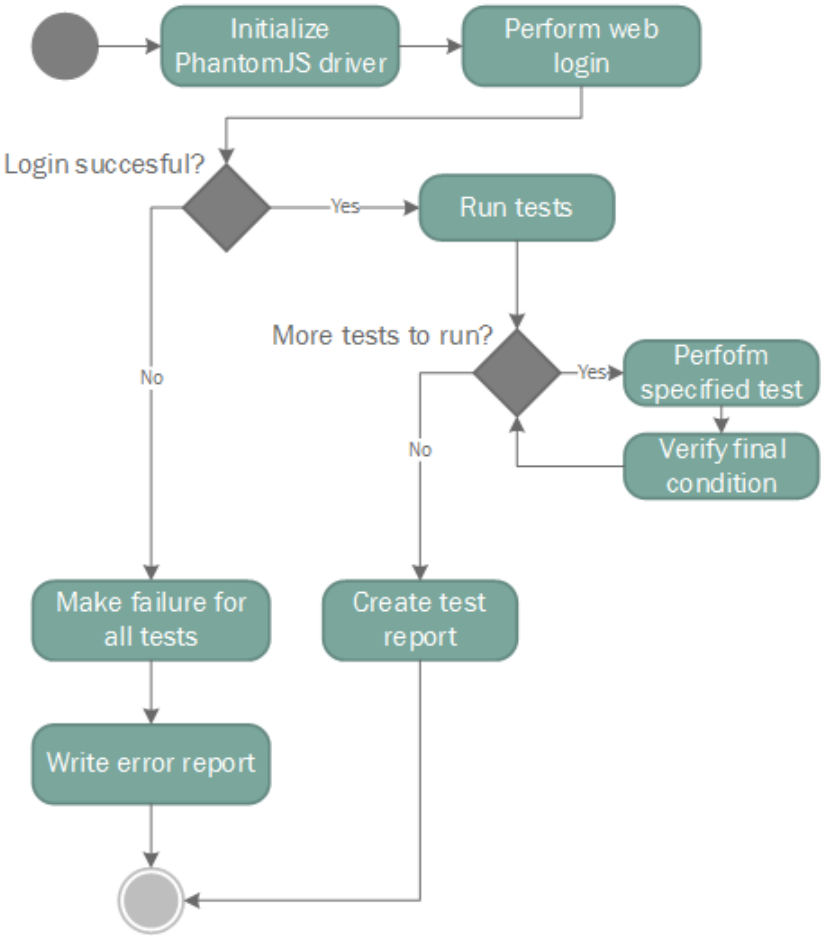
5.2.4 Návrh a Implementace UI testů

Poslední typ testů jsou testy simulující akce uživatele na webovém rozhraní systému GX-GO. Pro tento typ testů bude využita knihovna Selenium WebDrive a její rozšíření PhantomJS Drive. Pro normální testování přes Selenium je jeho funkcionalita napojena na webový prohlížeč a je relativně jedno na který, Selenium obsahuje ovládací rozhraní pro všechny běžné prohlížeče, jako Google Chrome, Mozzila Firefox, Safari, Edge a další.

Na rozdíl od běžného Selenia, rozšíření PhantomJS nevyžaduje žádný prohlížeč, funguje tak, že celou strukturu webu si uloží do paměti a pracuje s ní přímo bez nutnosti používat skutečné rozhraní. Příklad toho, jak bude test pracovat, je uveden na obrázku 9.

Tento test tedy začíná inicializací PhantomJS, kterému se zadá adresa pro webový systém, nad kterým budou testy prováděny. V inicializační fáze také provede přihlášení uživatele k testování. Pokud se přihlášení povede, pomocná proměnná definující validní přihlášení se podle toho nastaví, aby všechny ostatní testy věděly, že mohou pokračovat ve své práci. V opačném případě, že by přihlášení selhalo, selžou automaticky všechny testy.

Pro princip seznámení s testováním řekněme, že přihlášení proběhlo v pořádku. V tom případě všechny testy automaticky pokračují na webovou adresu, která je přístupná po přihlášení uživatele. Toto přesměrování se provádí pro každý test, z důvodu že každý z testů operuje v jiné části systému.



Obrázek 9: UI test demo

Samotné testování už funguje stejně jako u běžného Selenia. Pomocí funkcí obsažených v knihovně Selenium je možné vyhledávat specifické elementy ve webovém systému, přistupovat do těchto elementů, předávat jim hodnoty vstupu a tak dále. Každý test je prakticky sekvence příkazů na práci s elementy, kde tyto příkazy simulují reálného uživatele.

Jakmile by se test měl nacházet ve finální fázi, je nutné zkontrolovat, že byla splněna závěrečná podmínka, což by v případě Selenia měla být například existence určitého elementu nebo přítomnost určité hodnoty někde na adrese, kde se test právě nachází. Pokud se tato podmínka povede zkontrolovat, test je vyhodnocen jako splněný a je možné pokračovat k dalšímu. Pro představu tohoto typu testování je uveden ukázkový kód ve výpise 20.

```
PhantomJSDriver driver;
[ClassInitialize]
public void TestInit()
{
  driver = new PhantomJSDriver();
  driver.Manage().Timeouts().ImplicitlyWait(new TimeSpan(0, 5, 0));
```

```
  driver.Navigate().GoToUrl(mainWebUrl);

  driver.FindElement(By.Id("customerNumber_I")).SendKeys("testComp");
  driver.FindElement(By.Id("userName_I")).SendKeys("testUser");
  driver.FindElement(By.Id("password_I")).SendKeys("testPass");

  wait.Until(ExpectedConditions.InvisibilityOfElementLocated(By.ClassName("
      loginBody")));
}
```

Výpis 20: UI test demo - segment 1

Plná verze zdrojového kódu k tomuto testu je dostupná v sekci přílohy B na straně 82. V příkladu uvedeném výše začíná testování metodou TestInit s atributem ClassInitialize, který určuje, že metoda proběhne pouze jednou, když je testovací třída poprvé inicializována. V této metodě probíhá přihlášení uživatele do systému. Důvodem, že přihlášení je právě v Init metodě je ten, že by bylo kontraproduktivní provádět nové přihlášení pro každý specifický test.

Nejdříve je tedy nutno inicializovat PhantomJS WebDriver. Jak již bylo řečeno, jakmile se driver inicializuje a provede se přihlášení, bude tento driver sdílen mezi všemi testy. Než je možné přejít k samotnému přihlašování, je nutné provést přesměrování na cílový web. K tomu slouží metoda .Navigate().GoToUrl, do které je jako vstupní parametr vložena adresa cílového webu. Jakmile se driver přesměruje na web, je třeba identifikovat tři konkrétní elementy.

Jde o elementy pro vložení identifikačního čísla, uživatelského jména a hesla. Po identifikování elementů je do nich třeba poslat přihlašovací údaje pomocí metody .SendKeys. Po odeslání parametrů pro přihlášení je třeba identifikovat tlačítko pro přihlášení a pomocí funkce .Click() simulovat jeho stisk. Potom už je jen třeba počkat, než proběhne přihlášení a je možné zkontrolovat podmínku, že element pro přihlášení zmizel.

```
[TestMethod]
public void WatchdogModifyTest()
{
 testDrive.Navigate().GoToUrl(loggedWebUrl);

 Actions action = new Actions(testDrive).MoveToElement(testDrive.FindElement(By
     .Id("HeaderMenu_DXI5_"))).Perform();

 wait.Until(ExpectedConditions.InvisibilityOfElementLocated(By.ClassName("
     LoadingPanel")));

 if (IsElementPresent(testDrive, By.Id("wdEventList_DXDataRow0")))
 {
```

```
  rightClick.MoveToElement(el);
  menu.FindElement(By.Id("aeEventPopup_DXI0_")).Click();
  var update = testDrive.FindElement(By.ClassName("eventUpdate"));
  Assert.IsTrue(update.Displayed);
 }
 else
 {
  Assert.Fail("Unable to find event for update testing");
 }
}
```

Výpis 21: UI test demo - segment 2

Po korektním provedení inicializační metody se dále provede testovací metoda WatchdogModifyTest. Jak je vidět, na úplném začátku metody se kontroluje proměnná canTestingContinue, která říká, jestli se povedlo přihlášení a tím pádem jestli je vůbec možné pokračovat v testování. Po ověření této podmínky, pokud je podmínka splněna, se přiřadí výchozí webDriver nové proměnné, která bude sloužit k testování. Dále je nutné provést přesměrování na adresu po přihlášení pomocí metody .Navigate().GoToUrl. V této testovací metodě se bude simulovat úprava události v modulu Watchdog.

Nejprve je tedy nutné se do modulu Watchdog dostat. Protože přistup k modulu vede přes menu, které reaguje na kurzor myši, aby se zobrazilo, je třeba definovat tuto akci pomocí třídy Actions. V inicializaci specifikujeme, pro který webDriver bude akce provedena. Dále pomocí parametru .MoveToElement je možné specifikovat, na který element se má simulovat akce kurzoru. Samotná akce se pro provedení ukončí parametrem .Perform(). Po této akci bude horní menu zobrazeno a bude tak možné kliknout na specifikovanou položku, v tomto případě položku Watchdog.

Po kliknutí je třeba počkat, než se modul načte a zmizí nahrávací panel. Pak je třeba zkontrolovat, že v levém panelu v seznamu událostí existuje nějaká událost, kterou by bylo možné upravovat. Pokud taková událost existuje, pomocí další inicializace třídy Actions se nasimuluje pravý klik na specifikovaný řádek v tabulce a ve výsledném kontextovém menu se zvolí položka pro úpravu události. Poté je opět nutno počkat, až dojde k načtení události a zmizí nahrávací panel. Jako finální parametr se kontroluje, jestli se na současné adrese objevily elementy indikující změnu události. Pokud jsou tyto elementy nalezeny, test je vyhodnocen jako splněný a pokračuje se dalším testem na seznamu.

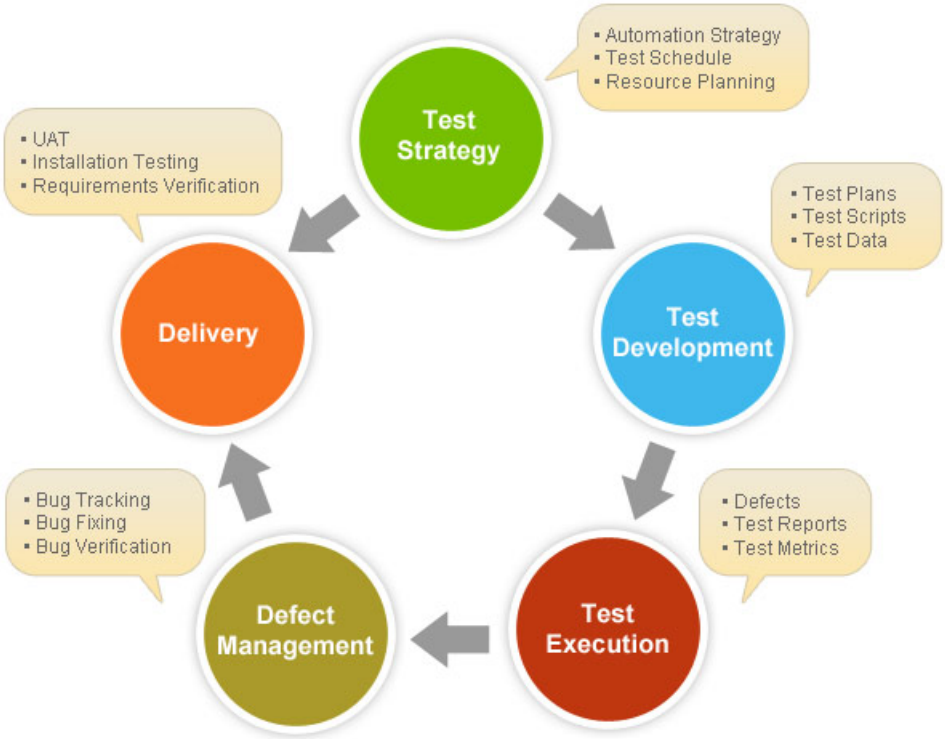
6 Testování

Tato sekce se bude věnovat testování modulu a to jak z pohledu funkcionálního, tak z pohledu výkonnostního. Oba testy modulu budou detailně rozebrány. U funkcionálních testů se bude kontrolovat hlavně výsledek, kterým je výpis výsledků testů v rámci jedné iterace modulu. Pro výkonnostní testy budou jako pomůcka sloužit grafy zatížení procesoru a operační paměti.

Funkcionální test kontroluje korektnost modulu jako celku, tedy připojení k *SVN* serveru, kompletní stažení nového repositáře nebo jen update stávajícího, překlad všech projektů s kontrolou korektnosti překladu, inicializace a konfigurace testů, korektní čtení výsledků testů a jejich zakomponování do celkového přehledu a nakonec samotné generování výsledné zprávy o testech.

Výkonnostní testy se budou primárně zaměřovat na vytížení hardware komponent v rámci běhu. Pro kontrolu výkonu budou sloužit grafy zobrazující vytížení zdrojů v rámci celé jedné iterace modulu.

V rámci testování se budou porovnávat dvě separátní iterace modulu. V rámci první iterace jde o první běh po integraci modulu to hlavního systému GX-GO. Druhá iterace ukazuje průběh o pět dní později a jak bude vidět.



Obrázek 10: Testovací cyklus, převzato z `http://iseb-softwaretesting.blogspot.cz/2013/11/test-management-tools.html`

6.1 Testování výsledků testů

Pro testování funkcionality modulu zde slouží kontrola samotných výsledků testů. V jednom iteračním cyklu jsou vždy testovány všechny testovací projekty, tedy za předpokladu, že od poslední iterace modulu došlo ke změnám na *SVN* serveru. V případě, že ke změnám nedošlo, testování ztrácí pointu z důvodu, že obdržené výsledky by byly stejné jako v předchozí iteraci a tím pádem by nepřinesly žádné nové poznatky.

První test výsledků je výstupem první iterace modulu po jeho integraci na GX-GO aplikační server.

Tabulka 2: **Iterace modulu - 9.4.2017**

	Testy				Počet	Výsledný
	Typ	Počet	Úspěšné	Neúspěšné	chyb	stav
ApplicationServer.Service	Unit	9	9	0	0	Pass
ApplicationServer.WCF	Unit	66	45	21	7	Fail
GX.ExternalAPI	Unit	3	3	0	0	Pass
ModuleAetr	Unit	2	2	0	0	Pass
ModuleDispatcher	Unit	1	1	0	0	Pass
ModuleDispatcherSheet	Unit	1	1	0	0	Pass
ModuleOptimizerOrders	Unit	1	1	0	0	Pass
ModuleReports	Unit	2	2	0	0	Pass
ModuleRouting	Unit	3	1	2	2	Fail
ModuleWatchdog	Unit	5	5	0	0	Pass
ProviderGPS.Aetr	Unit	2	2	0	0	Pass
ProviderGPS.DrivesCreator	Unit	9	7	2	1	Fail
ProviderGPS.Service	Unit	6	6	0	0	Pass
ProviderGPS.WatchDog	Unit	10	8	2	0	Fail
ProviderGPS.WCF	Unit	3	3	0	0	Pass
RepositoryMockTest	Mock	92	92	0	0	Pass
SeleniumUiTesting	UI	22	20	2	1	Fail
Shared.DB	Unit/Mock	124	122	2	2	Fail
Shared	Unit	9	9	0	0	Pass
WebApp	Unit/UI	8	3	5	2	Fail

Jak je vidět v tabulce výsledků 2, došlo zde k několika selháním v testech. Tato selhání měla různé příčiny. Jednou z příčin byl fakt, že testy samy o sobě byly chybné a vyvolaly výjimky v kódu. To ve finální fázi zapříčinilo selhání testu jako celku. Další poměrně jasná příčina byla nekorektní definice očekávání nebo výsledné hodnoty. To může být za některých

okolností požadovaný stav, nicméně v případě těchto testů jde o známku chyby.

Druhá kontrola výsledků proběhla na konci týdne, šest dní od první kontroly. Protože šlo o pracovní týden, v systému došlo k mnoha změnám.

Tabulka 3: **Iterace modulu - 15.4.2017**

	Testy				Počet	Výsledný
	Typ	Počet	Úspěšné	Neúspěšné	chyb	stav
ApplicationServer.Service	Unit	9	9	0	0	Pass
ApplicationServer.WCF	Unit	66	63	3	3	Fail
GX.ExternalAPI	Unit	3	3	0	0	Pass
ModuleAetr	Unit	2	2	0	0	Pass
ModuleDispatcher	Unit	1	1	0	0	Pass
ModuleDispatcherSheet	Unit	1	1	0	0	Pass
ModuleOptimizerOrders	Unit	1	1	0	0	Pass
ModuleReports	Unit	2	2	0	0	Pass
ModuleRouting	Unit	3	3	0	0	Pass
ModuleWatchdog	Unit	5	5	0	0	Pass
ProviderGPS.Aetr	Unit	2	2	0	0	Pass
ProviderGPS.DrivesCreator	Unit	9	9	0	0	Pass
ProviderGPS.Service	Unit	6	6	0	0	Pass
ProviderGPS.WatchDog	Unit	10	10	0	0	Pass
ProviderGPS.WCF	Unit	3	3	0	0	Pass
RepositoryMockTest	Mock	92	92	0	0	Pass
SeleniumUiTesting	UI	22	22	0	0	Pass
Shared.DB	Unit/Mock	124	123	1	1	Fail
Shared	Unit	9	9	0	0	Pass
WebApp	Unit/UI	8	8	0	0	Pass

Výsledek druhé kontroly, viz tabulka 3, se od prvního celkem odlišuje. Testovacích projektů, které korektně splnily svou práci je zde mnohem více. U těch projektů, kde došlo k selhání, šlo vždy o chybu v komunikaci. Příčinou chyby v komunikaci byl náhlý výpadek služby, se kterou testy komunikovaly. Výpadek sám o sobě neměl spojitost se samotnými testy.

6.2 Testování výkonu modulu

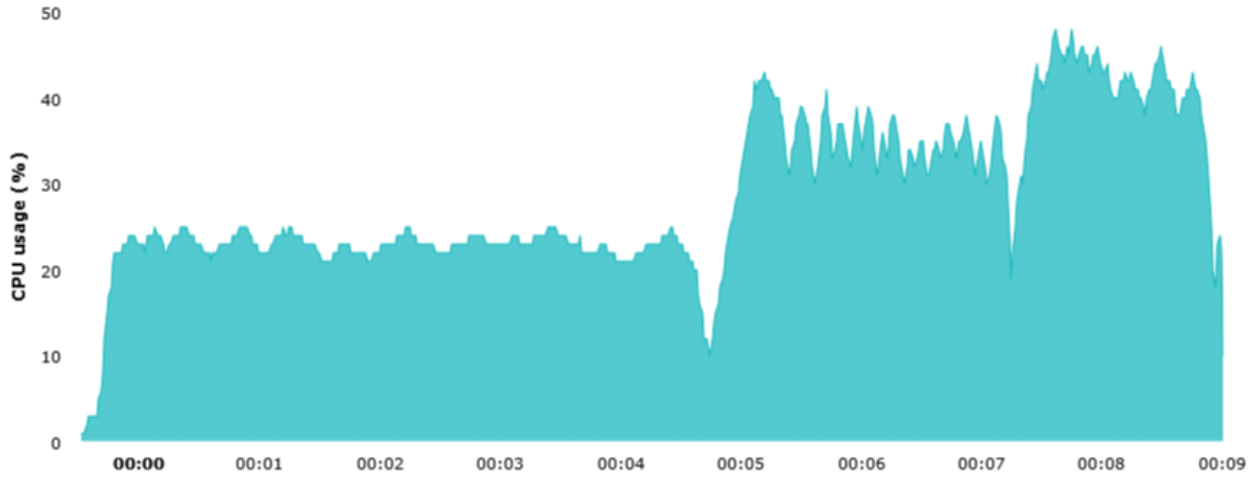
V sekci testování zatížení bude důraz kladen na využití fyzických komponent, jako je procesor a operační paměť. Kontrola zátěže je důležitá, aby se vedělo, kolik prostředků modul pro svůj běh přibližně požaduje.

Modul pro svou činnost využívá jedno jádro z procesoru. Toto jádro je sice sdílené sdílené s dalšími moduly, nicméně vzhledem k faktu, že modul automatizovaného testování běží v přesně definovaných a opakovaných iteracích jednou za den, není potřeba aby měl modul k jádru výhradní práva.

Byly zmíněny dvě komponenty, procesor a operační paměť. Pro lepší představu jsou níže uvedeny parametry zařízení, na kterém byl modul testován:

- IntelCore i5-2400 Sandy Bridge
- 3GB DDR3 SDRAM
- Gigabyte GA-B85-HD3
- Seagate BarraCuda 500GB HDD
- Corsair VS650 650W
- OS Windows 10

Pro porovnání jsou zde použity stejné dvě iterace, jako tomu bylo v předchozí sekci. První graf na obrázku 11 se zaměřuje na vytížení procesoru v rámci běhu modulu.



Obrázek 11: Zatížení CPU - test 1

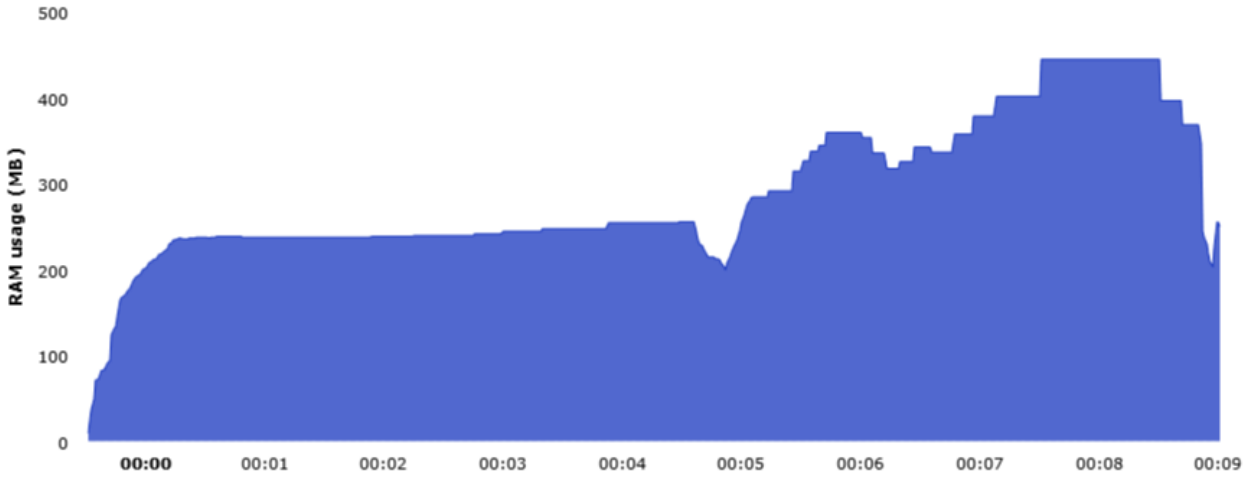
Celá iterace modulu při prvním spuštění trvala něco málo přes devět minut. Čas pro první iteraci je vždy mnohem vyšší něž pro jakoukoliv následující iteraci. Důvodem pro tento fakt je skutečnost, že při prvním spuštění je třeba na lokální úložiště stáhnout kopii vzdáleného repositáře. Tato samotná operace je časově náročná. V grafu zatížení *CPU* na obrázku 11, operace připojení k *SVN* serveru a stažení kopie na lokální úložiště, zabrala asi čtyři a půl minuty. Samotné stažení má určitou náročnost na procesor, jak je vidět na grafu na obrázku 11.

Po operaci stažení následuje překlad všech projektů a extrahování relevantních souborů pro testy. Překlad projektů trvá přibližně jedna a půl až dvě minuty, nicméně jeho náročnost je o něco vyžší. V nejvyžším bodě dosahuje vytížení přes 40 procent. Nejvyžší bod v překladu projektů značí hlavní jádro systému, které je samo o sobě velice rozsáhlé a komplexní a obsahuje mnoho referencí na pomocné služby a databáze. Tyto reference je samozřejmě zpracovat spolu s hlavním jádrem, proto je vytížení tak vysoké.

Další na řadě je samotné testování. Tedy použití již přeložených projektů z předchozího kroku, nakonfigurování parametrů pro testování a samotné provedení testů. Samotné testování je nejnáročnější část modulu. Hlavním důvodem je třeba simulace určité aktivity. Ze všech prováděných testů má nejvyšší náročnost headless testování za použití *PhantomJS*. Pro simulaci aktivity uživatele bez použití prohlížeče je proto třeba vytvořit virtuální prostředí, které obsahuje celou strukturu cílového webu. Celková doba testů pro tuto iteraci je asi minuta a půl, což je vzhledem k počtu testů a prováděným operacím dobrý výsledek.

Poslední část iterace je vygenerování konečného reportu na základě uložených údajů. Report se ukládá ve formátu *XML* a buduje se postupně ze všech testovaných projektů, jejichž výsledky byly viditelné v tabulce 3 na straně 59. Po vygenerování reportu iterace končí, všechna potřebná data jsou uložena a vypočítá se čas do další iterace.

Společně s kontrolou vytížení *CPU* za běhu modulu byl také kontrolován stav využití paměti RAM. Protože modul provádí mnoho operací s některé z nich vyžadují operační paměť v řádu stovek MB a navíc modul nebude na serveru sám, je třeba vědět, kolik bude tento modul průměrně spotřebovávat této paměti. K tomu pomůže graf na obrázku 12, který znázorňuje použití operační paměti ve stejném časovém horizontu jako graf na obrázku 11.



Obrázek 12: Zatížení RAM - test 1

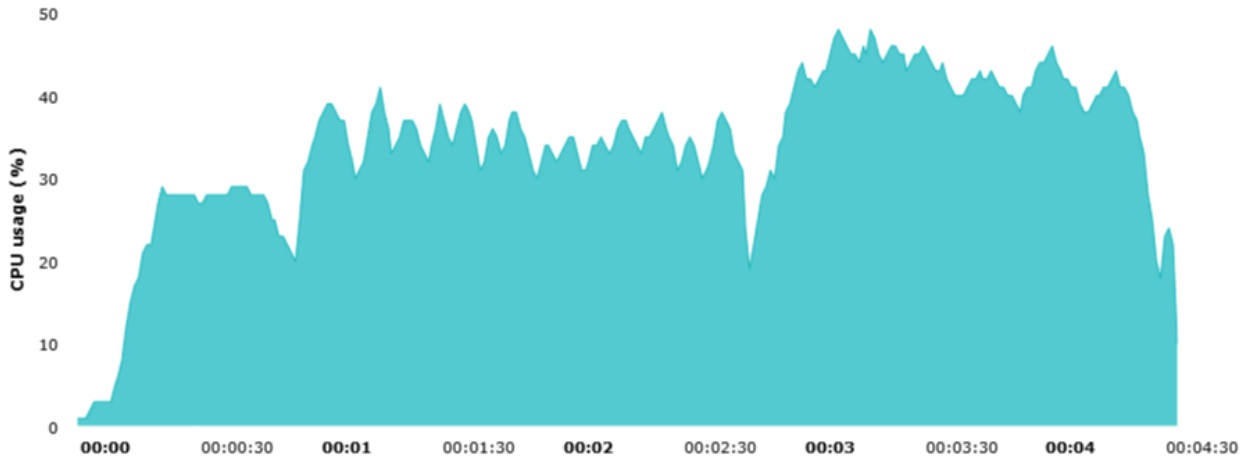
Jak je vidět, operace stažení lokální kopie z *SVN* není zas tak náročná, graf jasně znázorňuje velmi pomalý růst po počáteční alokaci paměti. I když tedy proces stažení trvá něco málo přes

čtyři minuty, stažená data jsou z operační paměti přesouvána na úložiště v přiměřené rychlosti, proto se operační paměť nepřenese přes 300MB.

V další fázi je opět vidět skok ve využití paměti, jde opět o operaci překladu projektů. Jak bylo zmíněno dříve, překlad bere v úvahu vazby mezi projekty a využité nástroje, proto je třeba více paměti aby projekty bylo možné bez problému zkompilovat a složit dohromady.

Největší skok je stejně jako ve vytížení procesoru registrován v části testování. Protože hlavně *PhantomJS* potřebuje velký obsah paměti kvůli headless simulaci webového prostředí, operační paměť v tomto bodu přesahuje hranici 400 MB. PhantomJS obsáhne ve svém prostředí jak kompletní responsivní strukturu webu, který by byl normálně viditelný uživateli, tak veškerá data, která s chodem systému souvisí.

Vygenerování reportu už zabýrá zanedbatelnou paměť vzhledem ke zbytku operací provaděných modulem automatizovaného testování. Nicméně, protože grafy na obrázcích 11 a 12 jsou validní pouze pro případ prvního spuštění modulu, není třeba jim v celkovém pohledu přikládat příliš velkou váhu. Poněkud relevantnější je následující dvojice grafů, které ukazují vytížení zdrojů v rámci dalších iterací. Tedy těch iterací modulu, kde modul již má stažený lokální repositář z *SVN* serveru a jediná další operace prováděná s *SVN* je update v případě, že je potřeba. Graf na obrázku 13 proto ukazuje zatížení procesoru v rámci běžného chodu modulu.



Obrázek 13: Zatížení CPU - test 2

Pokud tedy opomeneme operaci stažení lokální kopie z *SVN*, která byla časově nejnáročnější, celkový chod iterace modulu se pohybuje pod pěti minutami. Samotný graf je také poněkud detailnější, tento fakt je spojen se skutečností, že tato iterace zabírá mnohem méně času než iterace v předchozím příkladu. V tomto příkladu, operace update zabere asi dvacet vteřin, což je celkem odpovídající doba vzhledem k počtu provedených změn a přidaných objektů.

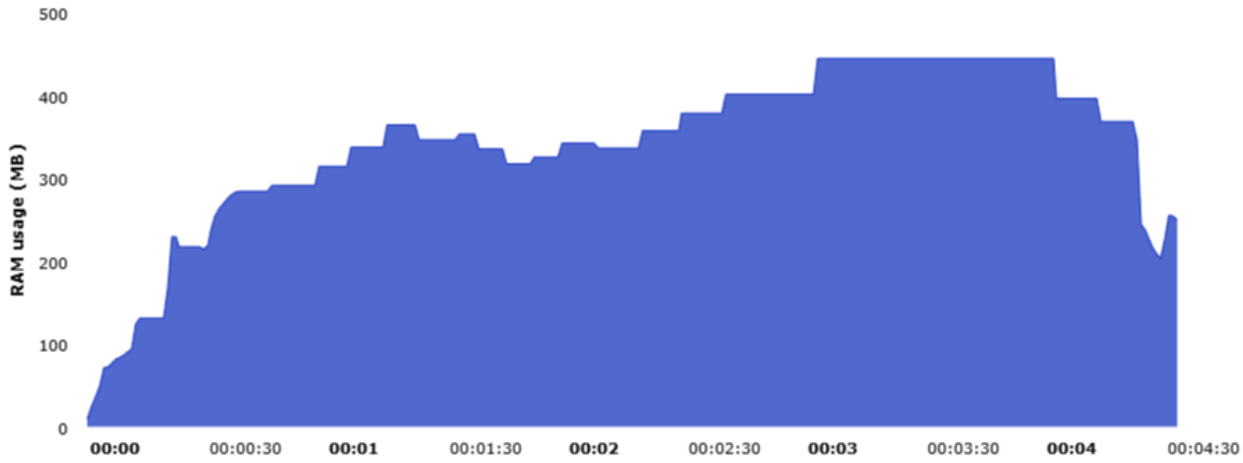
Operace překladu projektů zde zabírá víceméně stejný čas jako v grafu na obrázku 11, i když totožné nejsou. Reálně ani totožné být nemohou, každý chod je závislý mimo jiné na tom, kolik dalších procesů je právě zpracováváno. V tomto případě překlad projektů zabral asi minutu

a půl, rozdíl je ten, že vytížení procesoru nedosáhlo tak vysoko jako tomu bylo v předchozím případě.

Ve fázi testování je vidět několik vyšších a nižších míst, stejně tak jak tomu bylo u prvního příkladu. Celkové zatížení ale v součtu nedosahuje tak vysoko. Testování v tomto případu trvalo také asi minutu a půl, s určitou tolerancí. Ve vytížení procesoru je v době testování opět vidět ke konci velký skok, ten samotný značí *PhantomJS* test, který jak již bylo řečeno, je jedním z nejnáročnějších prováděných.

Finální operace zapsání výsledků testů do reportu není náročná, proto ani na grafu není vidět žádný výrazný skok. Výsledky druhého testu, viz tabulka 3 na straně 59, také neobsahují tolik chyb jako v prvním testovacím případu, viz tabulka 2 na straně 58. Jak bylo řečeno, chyby ve druhé testovací iteraci modulu automatizovaného testování byly způsobeny výpadkem s komunikační službě poskytující data.

Posledním grafem na obrázku 14 z této sekce je graf využití paměti RAM pro běžnou testovací iteraci. Tento graf se pohybuje opět ve stejném časovém horizontu jako graf na obrázku 13.



Obrázek 14: Zatížení RAM - test 2

Operace update, stejně tak jako u grafu zatížení procesoru, trvá stejnou dobu. Na grafu jsou vidět jisté nepravidelnosti, které jsou způsobeny právě velkým objemem dat a objektů přenášených ze serveru na lokální úložiště.

Hned po operaci update přichází operace překladu projektů, která zatížení paměti ještě zvedne. V nejvyšším bodě operace překladu čerpá skoro 400MB z operační paměti. I tak ale nejde o nějak závratné hodnoty, pokud vezmeme v úvahu, že modul běží v iteraci která je předem definovaná.

Testování je, stejně jako v případu na obrázku 12, strana 61, zabere operační paměti nejvíce. Opět, stejně jako v prvním případu, nejvíce paměti sebere *PhantomJS* kvůli případům, které již byly zmíněny, proto je netřeba opakovat.

Po vygenerování výsledného reportu iterace končí a bude čekat na další běh. Všeobecně druhá ukázaná iterace se nejvíce blíží realitě a ukazuje přibližné zatížení, jaké modul má na hardwarové komponenty.

6.3 Testování - souhrn

Testování mělo celkově prokázat několik věcí. Těmi hlavními bylo prokázat, že modul funguje správně a podle požadavků. Dále bylu nutné prokázat a otestovat, že modul nebude vytvářet přílišnou zátěž na systém jako celek. Prootže tento modul má být autonomní součást, která pracuje jen v dezignované časy, bylo třeba zaručit co nejvyžší efektivnost. Proto je modul v současné konfiguraci nastaven na aktivaci každý den o půlnoci. V tento čas totiž není zaznamenána žádná výrazná aktivita ze strany uživatelů, proto šance, že by chod modulu narušila jiná část systému, je zanedbatelná.

Z hlediska funkcionality modul splňuje všechny požadavky. Už bylo prokázáno, že pomohl odhalit několik nedostatků v jádru systému, právě díky automatickým testům, které testují moduly a korektnost cílových funkcí. Protože testy se v principu nemění, chyba v testu s jedné iteraci modulu, kde stejná chyba v předchoyích iteracích nebyla, značí narušení existující funkcionality.

Výkonnostní test měl za úkol prověřit, že modul nepřetěžuje zdroje systému. Protože modul reálně provádí testy nad systémem, kde v těchto testech se mohou vyskytnout operace jako komunikace s externími službami, dotazy na databázi nebo komplexní procesy v jádru systému, je třeba najít ideální bod, ve kterém by měl modul automatizovaného testování běžet. Po delší analýze chodu systému v rámci několika dní vyšel závěr, že nejefektivnější bude spouštět modul v době kolem půlnoci nebo v brzkých ranních hodinách. V této době je aktivita uživatelů absolutně minimální, jediné aktivity které nějak využívají zdroje systému jsou automatizované služby a operace zpracovávající pozicová data vozidel.

Právě z tohoto důvodu byl důležité výkonnostní testy, aby ukázaly, jak přibližně dlouho trvá jedna iterace modulu za běžných podmínek, pokud opomeneme první chod, ve kterém se zpracovává navíc stažení lokální kopie repositáře z *SVN*. Po několika testech byl vyvozen závěr, že iterace modulu zabere průměrně něco málo mezi pěti a sedmi minutami. Čas je mimo jiné závislý na současném vytížení.

7 Závěr

V první části tohoto dokumentu byla detailně rozebrána analýza pro modul Automatizovaného testování. Nejprve bylo nutné se podívat na modul z pohledu celého systému, tedy kam bude třeba modul zařadit a jak by měl komunikovat s ostatními částmi systému, pokud by to bylo potřeba. Dále také bylo rozebráno, jak má samotný modul pracovat, jaké bude využívat nástroje a jak budou fungovat jednotlivé testovací fáze.

Dále byly rozebrány různé komerční řešení v současné době dostupné na trhu. Každá z těchto technologií má jisté silné a slabé stránky, nicméně žádná nesplňuje všechny požadavky na testovací modul, jak je vidět v tabulce 1 na straně 23. Z toho důvodu bylo třeba pokračovat dále. V případě, že by nějaké komerční řešení vyhovovalo, nebyla by nutnost implementovat řešení vlastní.

Rozbor technologií se soustředil na popis nástrojů a knihoven, které budou třeba použít pro korektní požadovanou funkcionalitu modulu. V této části analýzy byly zmíněny knihovny jako *MOQ*, Selenium WebDriver, *PhantomJS*, *UnitTesting*, *SharpSVN* a jiné. Pro každou ze zmíněných knihoven bylo specifikováno, jakou funkcionalitu poskytují a jak by měly být použity v modulu, aby bylo dosaženo požadovaných výsledků.

V analytické fázi byl modul rozdělen na několik stěžejních částí, které bylo nutné řešit jako individuální komponenty, které ve finální fázi budou úzce spolupracovat za účelem korektního automatizovaného testování. Také bylo specifikováno, jak mají jednotlivé části modulu vypadat, jaké funkce budou plnit, jaké technologie budou využívat a jak budou komunikovat s dalšími částmi, pokud to je potřeba.

Část návrhu a implementace se zabývala už konkrétním návrhem jednotlivých částí modulu, stejně tak jako modulu jako celku. Bylo zde rozebráno, jak konkrétně budou jednotlivé části fungovat. Ukázkové kódy měly za úkol poskytnout detailnější náhled na to, jak vypadá samotná implementace. Kromě samotného modulu zde byly rozebrány i samotné testy a jejich typy, které jsou v systému používány. Stejně jako u modulu, všechny příklady testů byly doprovázeny diagramy a ukázkovými kódy.

Testovací část měla za úkol prokázat, že modul funguje přesně podle požadavků, že výsledky vracené iteracemi modulu vrací korektní výsledky a že tyto výsledky ve finální fázi pomohou v dalším vývoji a optimalizaci systému GX-GO. Mimo jiné se také povedlo ověřit, že modul pracuje efektivně a nepředstavuje nadměrnou zátěž na zbytek systému. Stejně tak se na základě analýzy běhu celého systému povedlo definovat ideální interval, ve kterém je nejlepší spouštět modul automatizovaného testování.

Nakonec je jen nutné říct, že modul Automatizovaného testování pro systém GX-GO byl vytvořen přesně podle požadavků, jeho analýza, návrh i implementace byly průběžně kontrolovány vedením projektu a veškerý feedback jak ze strany vedení tak ze strany vývoje byl prodiskutován a akceptován. Modul jako celek splnil očekávání a v současné době je aktivní součástí systému GX-GO a podává každodenní zprávy o průběhu testování.

Literatura

[1] TestRail [cit. 19.2.2017]. Zdroj: http://www.gurock.com/testrail.

[2] TFS [cit. 19.2.2017]. Zdroj: https://www.visualstudio.com/tfs.

[3] Segron [cit. 19.2.2017]. Zdroj: http://www.segron.com/our-services/test-automation.

[4] Telerik [cit. 19.2.2017]. Zdroj: http://www.telerik.com.

[5] Belatrix [cit. 19.2.2017]. Zdroj: http://www.belatrixsf.com.

[6] Rapise [cit. 19.2.2017]. Zdroj: http://www.inflectra.com/Rapise/.

[7] Testsigma [cit. 19.2.2017]. Zdroj: https://betalist.com/startups/testsigma.

[8] SharpSvn Tutorial [cit. 19.2.2017]. Zdroj: https://sharpsvn.open.collab.net/docs/walkthrough.htm.

[9] Unit Test. Microsoft MSDN [cit. 19.2.2017]. Zdroj: https://msdn.microsoft.com/en-us/library/hh694602.aspx.

[10] MOQ for .NET [cit. 19.2.2017]. Zdroj: https://github.com/Moq/moq4/wiki/Quickstart.

[11] Selenium WebDriver for .NET [cit. 19.2.2017]. Zdroj: http://www.seleniumhq.org/docs/03_webdriver.jsp.

[12] PhantomJS wrapper for .NET [cit. 19.2.2017]. Zdroj: https://www.nrecosite.com/phantomjs_wrapper_net.aspx.

[13] Creating Automated Tests [cit. 19.2.2017]. Zdroj: https://msdn.microsoft.com/en-us/library/dd380755.aspx.

[14] Get started with continuous testing [cit. 19.2.2017]. Zdroj: https://msdn.microsoft.com/library/ms253138.aspx.

[15] Configure unit tests by using a .runsettings file [cit. 19.2.2017]. Zdroj: https://msdn.microsoft.com/en-us/library/jj635153.aspx.

[16] Run automated tests using tcm [cit. 19.2.2017]. Zdroj: https://msdn.microsoft.com/en-us/library/dd465192.aspx.

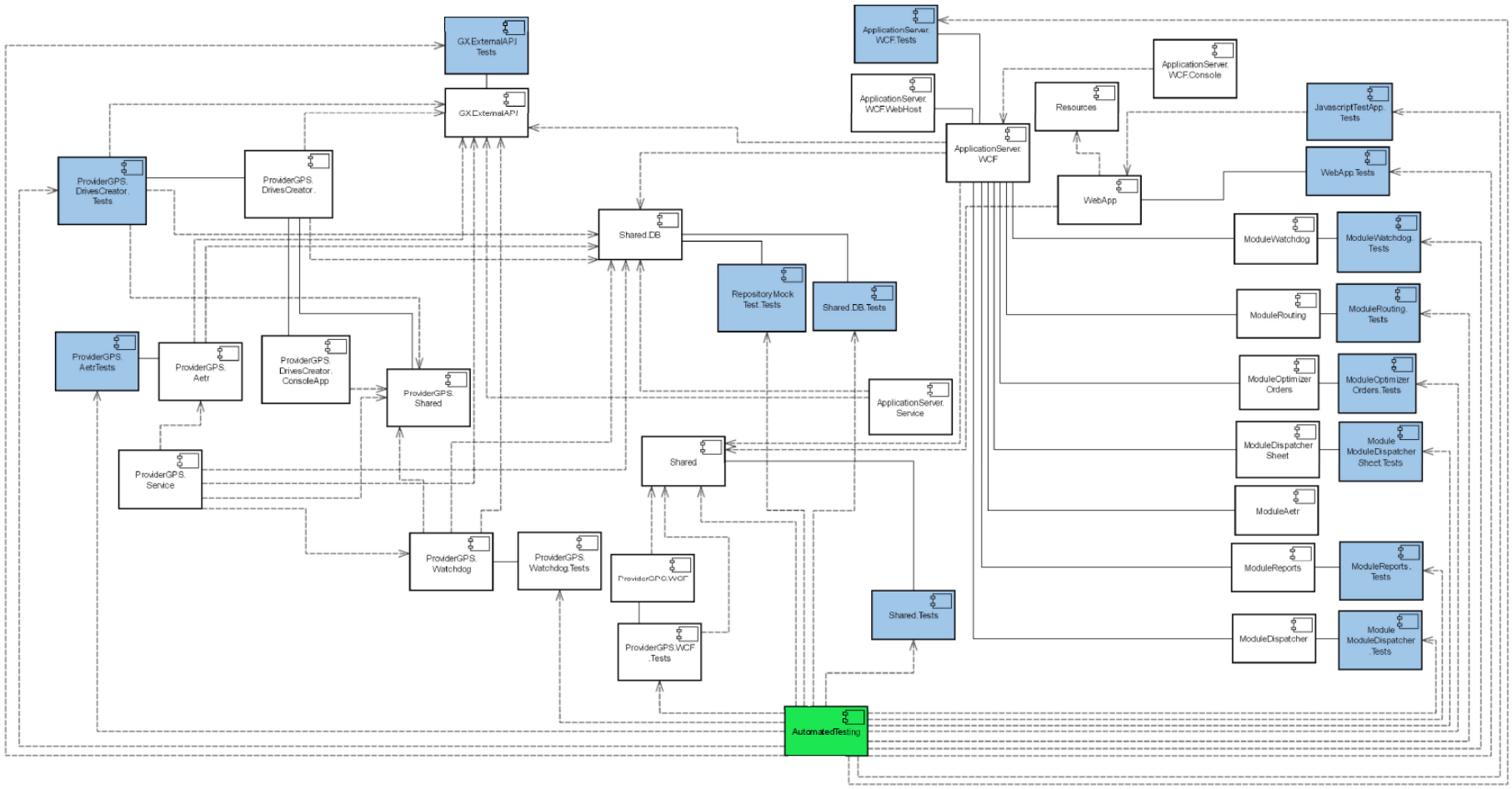
[17] ProcessStartInfo [cit. 19.2.2017]. Zdroj: https://msdn.microsoft.com/en-us/library/system.diagnostics.processstartinfo.aspx.

[18] Kent Beck, *Test Driven Development: By Example*, Three Rivers Institute, 2002.

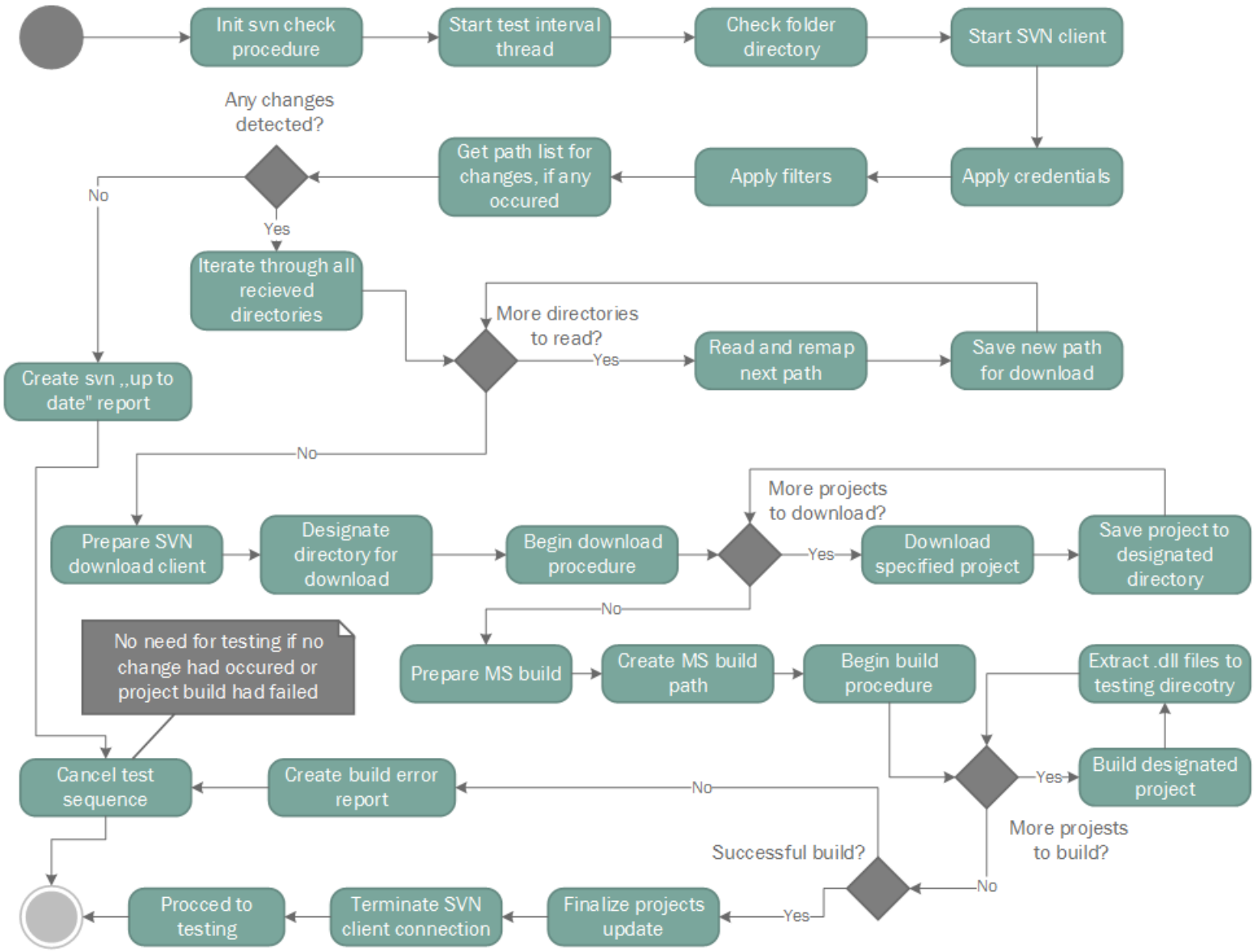
[19] Mark Fewster, *Experiences of Test Automation: Case Studies of Software Test Automation*, Addison-Wesley Professional, 2012.

[20] Elfriede Dustin, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*, Addison-Wesley Professional, 2009.

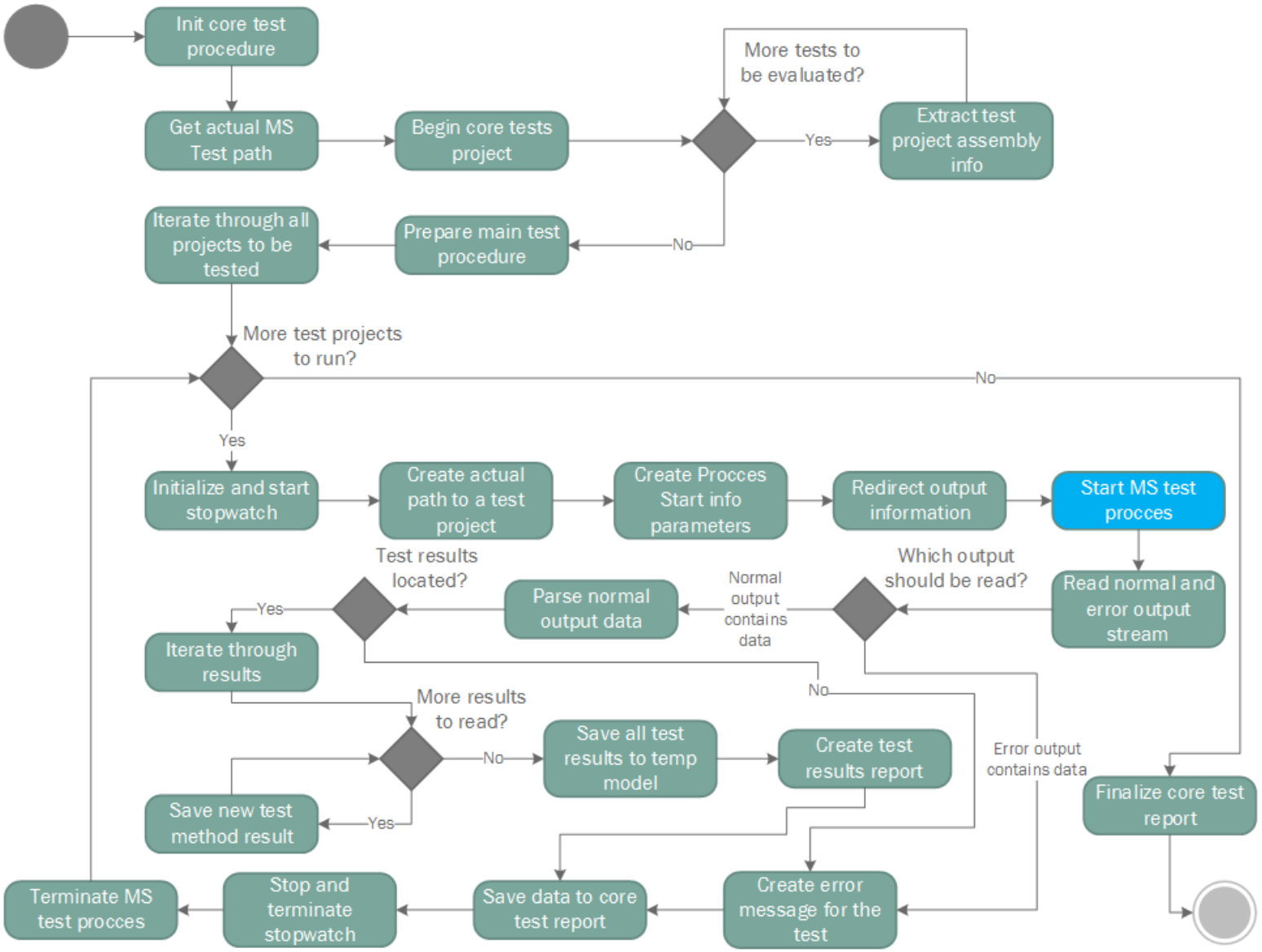
A Diagramy



Obrázek 15: Komponenty systému



Obrázek 16: Kontrola SVN a zpracování projektů - plná verze



Obrázek 17: Testování projektů - plná verze

B Návrh a Implementace - celé kódy

```
using (SvnClient svn = new SvnClient())
{
  svn.Authentication.ForceCredentials(USER, PSWD);
  svn.Authentication.SslServerTrustHandlers +=
  delegate(object sender, SvnSslServerTrustEventArgs e)
  {
    e.AcceptedFailures = e.Failures;
    e.Save = true;
  };
  svn.List(new Uri(SVN_URL), new SvnListArgs()
  {
    Depth = SvnDepth.Infinity
  },
  delegate(object sender, SvnListEventArgs e)
   {
    if (e.Path.EndsWith(".Tests", StringComparison.OrdinalIgnoreCase) )
    filesToProcess.Add(e.Uri);
   }
  );
   if (filesToProcess.Count > 0)
   {
     svn.Update(path);
     return true;
   }
   else
   {
     ReportHelper.AddStatus(
     String.Format("{0}|{1}",
      "Test modules are up to date",
      DateTime.Now)
     );
    return false;
   }
}
```

Výpis 22: SVN update - celé

```
Directory.CreateDirectory(path);

using (SvnClient svn = new SvnClient())
{
  try
  {
    SvnInfoEventArgs info;
    Uri repos = new Uri(SVN_URL);

    svn.Authentication.ForceCredentials(USER, PSWD);
    svn.Authentication.SslServerTrustHandlers +=
    delegate(object sender, SvnSslServerTrustEventArgs e)
    {
     e.AcceptedFailures = e.Failures;
     e.Save = true;
    };

    svn.GetInfo(repos, out info);

    svn.CheckOut(repos, path);
    firstTimeRun = true;
    return true;
  }
  catch (Exception ex)
  {
    //write error report
    ReportHelper.AddError(
     String.Format("{0}|{1}|{2}",
     "Trunk checkout has failed:",
     DateTime.Now,
     ex.Message)
    );
    return false;
  }
}
```

Výpis 23: SVN checkout - celé

```
Directory.CreateDirectory(Settings.Default.BuildPath);

for(int i = 0; i < projectsToBuild.Length; i++)
{
 bool canPublish = true;
 var filename = Path.GetFileName(projectsToBuild[i]);
 var filenameClean = Path.GetFileNameWithoutExtension(projectsToBuild[i]);

 ProcessStartInfo startInfo = new ProcessStartInfo()
 {
   CreateNoWindow = false,
   UseShellExecute = false,
   FileName = Path.Combine(
   Environment.GetFolderPath(
   Environment.SpecialFolder.Windows),
   PathHelper.BUILD_CMD) + "MSBuild.exe",
   Arguments = PathHelper.BUILD_PARAMS.Replace("{0}", projectsToBuild[i]),
   WorkingDirectory = path,
   RedirectStandardOutput = true,
   WindowStyle = ProcessWindowStyle.Hidden
 };

 try
 {
  using (Process exeProcess = Process.Start(startInfo))
  {
   while (!exeProcess.StandardOutput.EndOfStream)
   {
     string line = exeProcess.StandardOutput.ReadLine();
     if (line.Contains("Error "))
     {
      canPublish = false;
     }
   }
   exeProcess.WaitForExit();
  }
 }
```

```
 catch (Exception ex)
 {
   //build failure
   buildFailure = true;
   return;
 }

 if (canPublish)
 {
   var folderPath = projectsToBuild[i].Split('\\');

   var trimmedPath = "";

   for(int j = 0; j < folderPath.Length - 1; j++)
   {
     trimmedPath = String.Format("{0}{1}\\", trimmedPath, folderPath[j]);
   }

   string pathBinDebug = Path.Combine(trimmedPath, @"bin\Debug");
   string pathWCF = Path.Combine(Settings.Default.BuildPath, filenameClean, "
       bin");
   DirectoryCopy(pathBinDebug, pathWCF, true);
   if (Directory.Exists(Path.Combine(trimmedPath, @"obj")))
   {
     Directory.Delete(Path.Combine(trimmedPath, @"obj"), true);
   }

   if (projectsToBuild[i].IndexOf(
    ".Tests",
   StringComparison.InvariantCulture) > 0)
   {
     EvalProjectToTest(trimmedPath, filenameClean);
   }
 }
}
```

Výpis 24: Project build - celé

```
Stopwatch watch = new Stopwatch();
string testDllFolder = new FileInfo(value).DirectoryName;
var time = DateTime.Now.GetHashCode();
var currentTest = key;
string dllPath = Path.Combine(value, "bin", String.Format("{0}.dll",currentTest
    ));

var currentResult = String.Format("testResultFile_{0}_{1}.trx", currentTest,
    time );

watch.Start();

Process myProcess = new Process();
ProcessStartInfo myProcessStartInfo = new ProcessStartInfo()
{
  WindowStyle = ProcessWindowStyle.Normal,
  UseShellExecute = false,
  WorkingDirectory = testDllFolder,
  RedirectStandardError = true,
  RedirectStandardOutput = true,
  FileName = testPath.Trim(),
  Arguments = "/testcontainer:" + dllPath + " /resultsfile:" + currentResult
};
myProcess.StartInfo = myProcessStartInfo;
myProcess.Start();

string outputErrors = myProcess.StandardError.ReadToEnd();
string output = myProcess.StandardOutput.ReadToEnd();

watch.Stop();
model.TimeElapsed = TimeSpan.FromMilliseconds(watch.ElapsedMilliseconds);

//here evaluate outputs and write them to model
ReportHelper.FinalizeTestModel(output, outputErrors, model);
myProcess.WaitForExit();
myProcess.Close();
```

Výpis 25: Project test - celé

```
XmlElement rep = parentDoc.CreateElement(
 String.Empty,
"Reports",
 String.Empty
);

var info = new CultureInfo("en-US");
var format = @"hh\:mm\:ss";

foreach(var data in list)
{
  XmlElement report = parentDoc.CreateElement(String.Empty, "Report", String.
      Empty);

  XmlElement name = parentDoc.CreateElement(null, "Project_name", null);
  XmlText value1 = parentDoc.CreateTextNode(data.Name);
  name.AppendChild(value1);
  report.AppendChild(name);

  XmlElement state = parentDoc.CreateElement(String.Empty, "Test_state", String
      .Empty);
  XmlText value2 = parentDoc.CreateTextNode(data.State);
  state.AppendChild(value2);
  report.AppendChild(state);

  XmlElement time = parentDoc.CreateElement(String.Empty, "Elapsed_time",
      String.Empty);
  XmlText value3 = parentDoc.CreateTextNode(data.TimeElapsed.ToString(format,
      info));
  time.AppendChild(value3);
  report.AppendChild(time);

  if(data.TestCaseList.Count > 0)
  {
   XmlElement methods = parentDoc.CreateElement(String.Empty, "Method_results",
       String.Empty);
```

```
   foreach(var method in data.TestCaseList)
   {
    XmlElement m = parentDoc.CreateElement(String.Empty, "Method", String.Empty
        );

    XmlElement mName = parentDoc.CreateElement(String.Empty, "Name", String.
        Empty);
    XmlText mNameText = parentDoc.CreateTextNode(method.Name);
    mName.AppendChild(mNameText);
    m.AppendChild(mName);

    XmlElement mRes = parentDoc.CreateElement(String.Empty, "Result", String.
        Empty);
    XmlText mResText = parentDoc.CreateTextNode(method.Result);
    mRes.AppendChild(mResText);
    m.AppendChild(mRes);

    methods.AppendChild(m);
   }

   report.AppendChild(methods);
  }
  else
  {
   XmlElement error = parentDoc.CreateElement( String.Empty,"Error_message",
       String.Empty);
   XmlText errotText = parentDoc.CreateTextNode(data.Error);
   error.AppendChild(errotText);
   report.AppendChild(error);
  }

  rep.AppendChild(report);
}
parentDoc.AppendChild(rep);
```

Výpis 26: XML document create - celé

```
[TestClass]
public class ReportCoreTest
{
 [ClassInitialize]
 public static void Initialize(TestContext testContext)
 {
  AutoMapper.Mapper.Initialize(config =>
  config.AddProfile<Shared.DB.Helpers.AutoMapperProfile>());
 }

 [TestMethod]
 public void GetReportDataMileageTest()
 {
  List<int> ids = new List<int>();
  ids.Add(3);
  ids.Add(4);
  ids.Add(10);
  ids.Add(15);
  ids.Add(30);
  ids.Add(31);
  ids.Add(32);
  ids.Add(33);
  ids.Add(34);
  ids.Add(35);

  var result = ReportsCore.GetReportDataMileage(ids,
   new AppContext()
   {
    CustomerID = 1,
    Language = 1029,
    UserID = 1
   }
  );

  Assert.IsTrue(result.Count != 0);
 }
}
```

Výpis 27: Core test demo - celé

```
[TestClass]
public class DeviceStoreTests
{
 [ClassInitialize]
 public static void Initialize(TestContext testContext)
 {
  DbInterception.Add(EFSimpleDatabaseLogger.Debug);
 }

 [TestMethod]
 public void GetDeviceProfileInputTest()
 {
  var appContext = new DataContracts.AppContext()
  {
    CustomerID = 1, UserID = 1,
  };
  var iut = new DeviceStore { AppContext = appContext };

  var result = iut.GetDeviceProfileInput(3);
  Assert.IsTrue(result != null);
 }

 [TestMethod]
 public void CreateTest()
 {
  var appContext = new DataContracts.AppContext() {CustomerID = 1, UserID = 1
      };

  var iut = new DeviceStore { AppContext = appContext };
  var result = iut.Create(new Repository.Device()
  {
    DeviceID = 95, IsDeleted = false, DeviceType = 1
  }, appContext.UserID);

  Assert.IsTrue(result != null);
 }
}
```

Výpis 28: Database test demo - celé

```
[TestClass]
public class UnitTest1
{
 private IDataStoreFactory realDataStore;
 private Mock<IDataStoreFactory> mockDataStore;
 private AppContext appContext;

 [TestInitialize]
 public void TestInit()
 {
  appContext = new AppContext() { CustomerID = 1, Language = 1029, UserID = 1
      };
  mockDataStore = new Mock<IDataStoreFactory>() { CallBase = true, DefaultValue
       = DefaultValue.Mock};
 }

 [TestMethod]
 public void ApiSourceStoreGet()
 {
  mockDataStore.Setup( s => s.GetDataStore<IApiSourceStore>( It.IsAny<
      AppContext>())).Returns<ApiSourceStore>(store =>
  {
   var mock = new Mock<IApiSourceStore>(){ CallBase = true };
   mock.Object.AppContext = appContext;
   mock.Setup(s => s.Get(It.IsAny<int>())).Returns(new ApiSource());
   return mock.Object;
  });

  var mockObject = mockDataStore.Object.GetDataStore<IApiSourceStore>(null);
  var result = mockObject.Get(1);
  Assert.IsInstanceOfType(result, typeof(ApiSource));
 }
}
```

Výpis 29: Mock test demo - celé

```
[TestClass]
public class UnitTest1
{
 PhantomJSDriver driver;
 private readonly string mainWebUrl = "";
 private readonly string loggedWebUrl = "";

 private bool canTestingContinue = true;
 private readonly string deviceNum = "5T9 3499";

 [ClassInitialize]
 public void TestInit()
 {
  try
  {
   driver = new PhantomJSDriver();
   driver.Manage().Timeouts().ImplicitlyWait(new TimeSpan(0, 5, 0));
   driver.Navigate().GoToUrl(mainWebUrl);

   var company = driver.FindElement(By.Id("customerNumber_I"));
   var username = driver.FindElement(By.Id("userName_I"));
   var password = driver.FindElement(By.Id("password_I"));

   company.SendKeys("testComp");
   username.SendKeys("testUser");
   password.SendKeys("testPass");

   driver.FindElement(By.Id("btnLogOn_I")).Click();

   var wait = new WebDriverWait(driver, TimeSpan.FromSeconds(150));
   wait.Until(ExpectedConditions.InvisibilityOfElementLocated(By.ClassName("
       loginBody")));
  }
  catch(Exception e)
  {
   canTestingContinue = false;
  }
 }
```

```
[TestMethod]
public void WatchdogModifyTest()
{
 if(canTestingContinue)
 {
  try
  {
   var testDrive = driver;
   testDrive.Navigate().GoToUrl(loggedWebUrl);
   new Actions(testDrive).MoveToElement(testDrive.FindElement(By.Id("
       HeaderMenu_DXI5_"))).Perform();
   testDrive.FindElement(By.Id("HeaderMenu_DXI5i2_")).Click();

   if (IsElementPresent(testDrive, By.Id("wdEventList_DXDataRow0")))
   {
    new Actions(testDrive).MoveToElement(el).ContextClick(el).Build().Perform
        ();
    var el = testDrive.FindElement(By.Id("wdEventList_DXDataRow0"));
    var menu = testDrive.FindElement(By.Id("aeEventPopup_DXME_"));
    menu.FindElement(By.Id("aeEventPopup_DXI0_")).Click();

    var update = testDrive.FindElement(By.ClassName("eventUpdate"));
    Assert.IsTrue(update.Displayed);
   }
   else
   {
    Assert.Fail("Unable to find event for update testing");
   }
  }
  catch (Exception e)
  {
    Assert.Fail(e.Message);
  }
 }
}
}
```

Výpis 30: UI test demo - celé

C Obsah CD

- Zdrojové kódy modulu
- Ukázky testů
- Dokumentace pro vývojáře GX
- Prezentace